MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un langage de haut niveau de manipulation de bases de données

Delvaux, Yves

Award date:
1977

Awarding institution:
Universite de Namur

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

ANNEE ACADEMIQUE 1976-1977

Implémentation d'un langage de haut niveau de manipulation de bases de données

YVES DELVAUX

Mémoire
présenté en vue de l'obtention
du grade de
Licencié et Maître en Informatique

REMERCIEMENTS

Je tiens à exprimer ma profonde gratitude à Jean-Luc Hainaut pour l'intérêt porté à ce mémoire et l'aide constante qu'il m'a prodiguée lors de la rédaction. Ses nombreux conseils m'ont permis de clarifier les difficultés théoriques de cette étude et d'en réaliser une implémentation partielle.

Qu'il me soit en outre permis de remercier l'équipe "Grands Fichiers" pour sa collaboration aussi sympathique qu'efficace durant mon travail.

TABLE DES MATIERES

CHAPITRE 2 : Etude du langage de manipulation

°
° °

CHAPITRE 3 : Le DML de l'Institut d'Informatique de Namur

°
° °

CHAPITRE 4 : Comparaison des principaux DML

o
o o

CHAPITRE 5 : Le compilateur du LDA

INTRODUCTION

Il est sans doute superflu d'évoquer l'essor qu'a pris, ces dernières années, l'emploi de bases de données, tant dans les entreprises que dans les pouvoirs publics.
Une base de données centralise des informations qu'il faut pouvoir décrire avant de les manipuler. Ce double rôle est assigné aux langages de bases de données, qui comportent généralement deux parties :

- un langage de définition de données permet de définir des données dont les types sont décrits dans un modèle de données;
- un langage de manipulation de données est un ensemble de primitives qui agissent sur les types de données du modèle.

Nous exposons dans ce mémoire, un langage de manipulation de données, qui sont décrites à l'aide d'un modèle défini par les chercheurs de l'équipe "Grands Fichiers" des FNDP. Ce langage, le LDA, est un langage de haut niveau, du type "langage de boucles", qui offre le moyen de rédiger des programmes d'exploitation structurés en boucles d'accès aux unités d'information.
Le but de ce mémoire est d'en réaliser une implémentation, qui réponde au double objectif suivant :

- à partir de programmes sources écrits en LDA, générer des programmes objets rédigés dans un autre langage de manipulation déjà implémenté à l'Institut, le DML;
- définir en outre un code intermédiaire entre les programmes sources et les programmes objets, qui soit suffisamment souple pour être réorganisé par le biais de modules d'optimisation; c'est pourquoi nous avons adopté un code intermédiaire possédant une structure arborescente.

La première partie de cet ouvrage rappelle les fondements du modèle de données, du type relationnel binaire, qui permet de superposer la structure sémantique et la structure d'accès des informations représentatives d'un système réel.
Dans la deuxième partie, nous présentons le langage LDA. Il était indispensable de rappeler brièvement les caractéristiques du langage "cible", le DML; c'est pourquoi nous décrivons ce dernier dans la troisième partie.
La quatrième partie est un exposé de certains langages de manipulation tels que SOCRATE, SEQUEL, IDS1 et IMS; nous nous sommes efforcés de comparer ces langages au LDA, sous un aspect formel.

Enfin, la cinquième partie livre les principes de l'implémentation du LDA.

Dans un premier temps, nous définissons le code intermédiaire et nous montrons de quelle façon il est construit.

En second lieu, nous exposons la génération du code objet, à partir du code intermédiaire.

°
° °

CHAPITRE 1 : Le modèle de structure d'information

1.1. Introduction

La majorité des processus d'analyse devant conduire à une réalisation comportent une première phase dont le rôle est de déterminer le domaine d'application de la réalisation. Dans le cadre d'une analyse en termes de bases de données, cette phase fonctionnelle mène à la définition de la structure conceptuelle de l'information. Ce niveau conceptuel décrit un ensemble d'informations et de propriétés sans souci d'efficacité de manipulation de ces informations.

Il faudra, dans une deuxième phase, déterminer les chemins d'accès aux données propres à assurer l'écriture de programmes d'application efficaces. Dans la mesure où l'on ne désire pas se restreindre aux outils de description que propose un système de gestion de bases de données particulier, il est nécessaire d'adopter pour ces deux niveaux, un modèle conceptuel (ou sémantique) et un modèle d'accès d'une base de données.

Nous présentons ici un formalisme unique qui exprime les concepts propres aux deux niveaux : le modèle de structure d'information (M.S.I.).

1.2. Le MSI vu comme modèle sémantique

1.2.1. Eléments du modèle

Ce modèle propose de structurer les données d'une BD d'une manière analogue à la structure du réel perçu; une perception possible d'un système réel est d'y voir des éléments en relation binaire les uns avec les autres. Si l'on associe à chaque élément du système une unité d'information, on peut alors représenter les associations entre éléments par des associations entre les unités d'information correspondantes.

Admettons en outre que certains éléments sont si simples qu'ils peuvent être décrits par une valeur ou une information élémentaire (comme un nom, un numéro), alors que d'autres exigent une description plus complexe (comme une personne, une faculté).

Si l'on observe enfin que les éléments et les associations de même nature constituent des classes (comme la classe des facultés, la classe des personnes qui professent dans une faculté), nous considérerons des classes d'unités d'informations ainsi que des classes d'associations entre ces unités; nous appelle-

rons OBJET une classe d'unités d'information et RELATION, une classe d'associations.
Un tel modèle permet donc de définir une base de données comme étant un ensemble d'objets et un ensemble de relations.

1.2.2. Définitions

- Nous donnerons un nom à chaque objet et à chaque relation; il est utile cependant d'accepter la chaîne vide comme nom pour certaines relations (telles que celles exprimant les associations d'appartenance) qui seront dites "sans nom".
- Notons R(A, B) la relation de nom R de l'objet de nom A, appelé objet origine, vers l'objet de nom B, appelé objet cible.
- On notera simplement (A, B) dans le cas où la relation n'a pas de nom.
- Pour signifier que a est une unité d'information du type A, nous écrirons $a \in A$ et nous dirons que a est une réalisation de A.
- De manière plus formelle $R(A, B) \subset \{(a, b) | a \in A \text{ et } b \in B\}$
  Si $(a, b) \in R(A, B)$, (a, b) sera dénommée une occurrence de R.

1.2.3. Propriétés des relations

- Si $(a, b) \in R$, cela signifie que R associe a et b, mais aussi que b peut être obtenue à partir de a, par un mécanisme logique associé à R. (Nous adoptons le point de vue d'un utilisateur ignorant la notion d'accès physique)
- On peut déclarer une relation S(B, A) comme étant l'inverse d'une autre R(A, B); cela signifie : $(a, b) \in R \Leftrightarrow (b, a) \in S$.
  Un exemple est le couple de relations employeur-employé.
- Associons quatre entiers i_R-j_R, k_R-l_R à chaque relation R(A, B), tels que à tout instant :
  - toute réalisation de A est associée par R à n réalisations de B avec
    $i_R \leqslant n \leqslant j_R$
  - toute réalisation de B est associée par R à m réalisations de A, avec
    $k_R \leqslant m \leqslant l_R$

Exemples : conjoint (PERSONNE, PERSONNE) : Ø-1, Ø-1
enfant (PERSONNE, PERSONNE) : Ø-2Ø, Ø-2
(PERSONNE, NOM) : 1-1, Ø-∞

- Il est possible de créer, de modifier, ou de supprimer une occurrence de relation.

1.2.4. Propriétés des objets

L'ensemble des objets d'une BD est partitionné en trois sous-ensembles.

- Les objets élémentaires (OE) correspondent à des informations élémentaires représentées par une valeur (nombre, chaîne de caractères, ...); on suppose que ces valeurs sont données à priori ("un âge varie entre Ø et 1ØØ").

  Il n'est pas possible de modifier l'ensemble des réalisations d'un OE.

  Certains OE ont des réalisations constituées de la concaténation de plusieurs valeurs d'autres OE : ces premiers OE sont dits composés de ces derniers.

- Les objets complexes (OC) sont associés à des unités d'information moins simples, qui sont représentées par leurs associations avec d'autres unités.

  Les opérations propres à ce type d'objet sont :
  - la création d'une réalisation;
  - la suppression d'une réalisation.

- L'objet racine correspond à une unité d'information particulière qu'est la base de données elle-même. Chaque BD contient un objet racine qui peut être l'origine de relations du type Ø-J, K-1 dont les cibles sont des OC et qui traduisent des accès en séquence à la totalité ou à une partie des réalisations de ces OC. Ceci afin de satisfaire un des besoins de l'utilisateur, qui est d'obtenir toutes les réalisations d'un type d'objet, et ce l'une après l'autre.

1.2.5. Représentation graphique

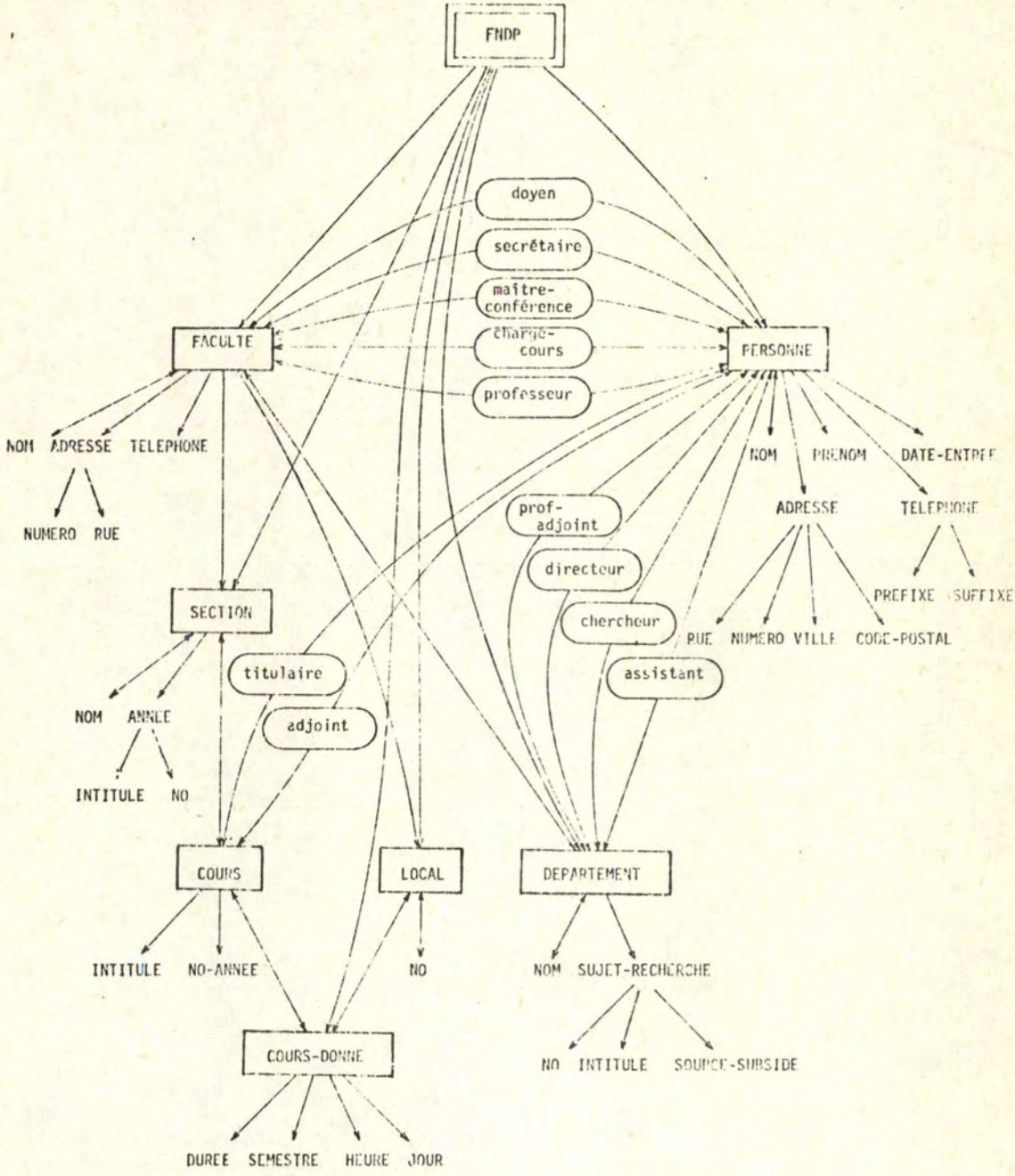
Il est souvent utile de représenter une structure de données par un diagramme. Nous conviendrons de figurer :

- une racine par un double rectangle contenant le nom de la BD;
- un OC par un rectangle contenant le nom de l'objet;
- un OE par son seul nom,
- une relation entre deux objets par un arc orienté entre les représentations des deux objets et éventuellement étiqueté par un cercle contenant le nom de la relation.

Si deux relations inverses l'une de l'autre portent le même nom, nous les représenterons par un seul arc orienté dans les deux sens.

1.2.6. Exemple

Voici le diagramme d'une base de données décrivant les Facultés de Namur.



- L'objet COURS-DONNE traduit le fait qu'un cours est donné plusieurs fois à des heures et dans des locaux différents;
- L'objet élémentaire ANNEE est composée des OE INTITULE et NO;
- Une description plus précise est disponible dans [1].

1.3. Le MSI vu comme modèle d'accès

Une structure d'accès est modélisée à partir des principes suivants :

- un objet est associé à un type d'unités d'informations enregistrées, accessibles individuellement et significatives pour un utilisateur;
- une relation est associée à un mécanisme (de données et/ou d'algorithmes) permettant d'accéder à des réalisations d'un objet à partir d'une réalisation d'un objet (qui peut être identique) et ce, d'une manière qui reste significative pour un utilisateur.

1.4. Correspondance des deux modèles

1.4.1. Introduction

Nous pensons que ce formalisme est apte à soutenir la démarche classique d'analyse dite "par les données".

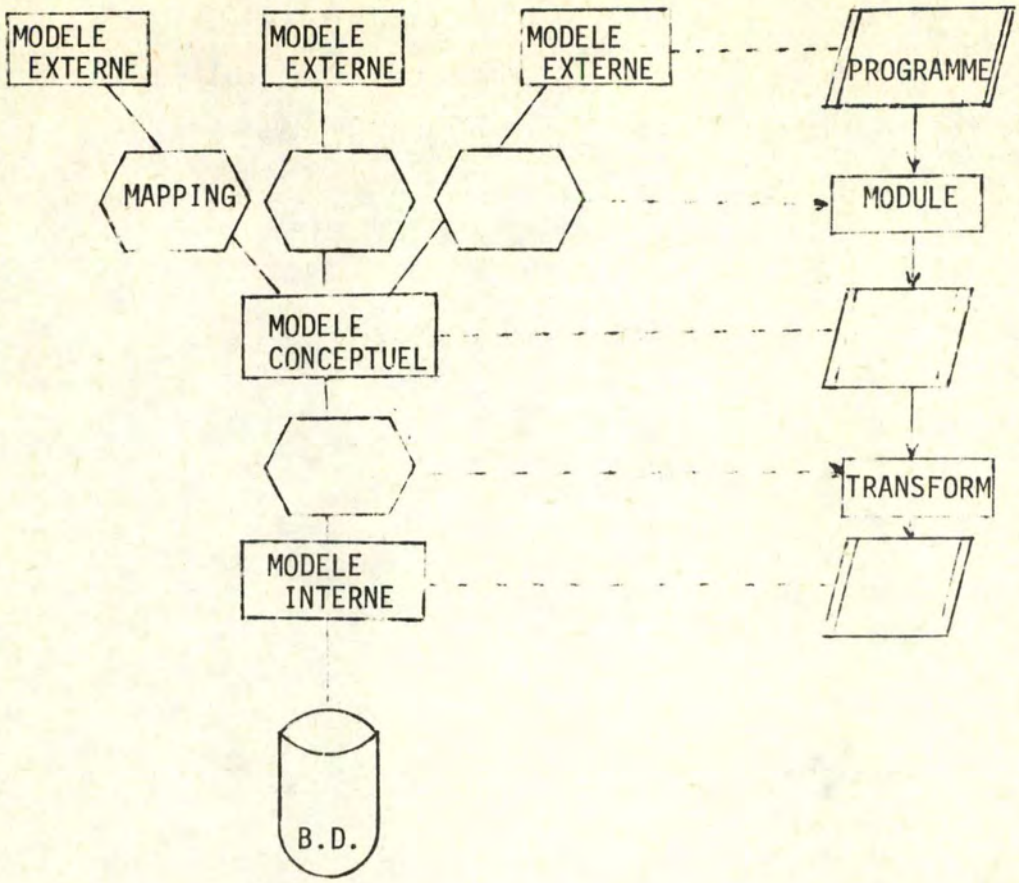
Dans un premier temps, l'utilisateur dégage les entités du système réel auquel il est confronté ainsi que leurs propriétés (comme leurs liens communs); le concepteur repère ainsi des objets et des relations, de manière très intuitive au départ; une approche plus détaillée permet de scinder l'ensemble des objets en objets complexes et élémentaires.

La deuxième étape vise à choisir les chemins d'accès optimaux aux unités d'informations en égard à la liste des problèmes à résoudre.

Il nous faut voir comment ces niveaux d'analyse sont emboîtés et nous allons en profiter pour élargir notre point de vue, en examinant brièvement une proposition parmi les travaux entrepris à ce sujet.

1.4.2. Niveaux de description d'une BD

Nous avons ainsi ébauché le principe de "division du travail" de l'analyse en niveaux : un de ses objectifs est de rendre un niveau de description indépendant des critères pris en charge dans les niveaux inférieurs. Rappelons qu'une analyse conceptuelle des informations doit être indépendante de toute considération d'accès : ceci implique entre autres qu'une modification dans le choix des accès n'affecte pas la description du niveau conceptuel. Le comité x3 de l'ANSI a publié un rapport [3] dans lequel il propose une architecture de SGBD (Système de gestion de BD) basée sur 3 niveaux descriptifs de données, que l'on peut schématiser comme suit.



Un modèle correspond à un niveau de description; c'est l'ensemble des données qui représentent les entités du monde réel.

- Le modèle conceptuel (encore appelé fonctionnel ou logique par d'autres auteurs) est une description du monde réel qui se veut stable au cours du temps : une modification de ce niveau ne devrait être nécessaire que si l'organisation elle-même (ou sa perception) change.

  Le MSI tel que nous l'avons exposé permet de modéliser un niveau semblable sous la forme de ce que nous appelons un graphe de base sémantique (GBS); nous pensons avoir montré qu'un tel graphe peut constituer, dans sa première interprétation, une description sémantique des données constitutives d'une organisation mais nous ne prétendons pas qu'un GBS est exactement un modèle conceptuel, au sens d'ANSI/SPARC.

- Le modèle interne est la description du niveau d'implémentation des chemins d'accès; dans notre système, un premier volet de description à un tel niveau peut être formalisé en termes du MSI, par un graphe de base descripteur (GBD).

  A. On peut en effet convenir de représenter :
     - un type d'enregistrement par un OC;
     - un champ de données par un OE;
     - un chemin d'accès entre deux types d'enregistrement par une relation entre OC;
     - le fait qu'un champ de données appartient à un certain type d'enregistrement par une relation de l'objet complexe vers l'objet élémentaire;
     - un accès séquentiel par une relation de la racine à un OC;
     - un accès par clé par une relation d'un OE vers un OC.

  B. Un second volet est la description des fichiers qui contiennent les données décrivant le système; cette description reprend la composition des articles internes (ordre, type, codification, ... des champs internes), les organisations, les méthodes d'accès, l'emplacement en mémoire auxiliaire, ...

- Les modèles externes veulent refléter le pluralisme des utilisateurs; la variété des problèmes posés aux utilisateurs engendre en effet une variété de perceptions du réel : le gestionnaire d'une bibliothèque, par exemple, a d'autres besoins que les emprunteurs et privilégie en conséquence certaines informations ainsi que les moyens d'y accéder.

  Un modèle externe est donc une collection d'articles vue par une classe d'utilisateurs; on peut encore considérer que le MSI convient pour définir des descriptions à ce niveau : ces descriptions seraient appelées des graphes secondaires.

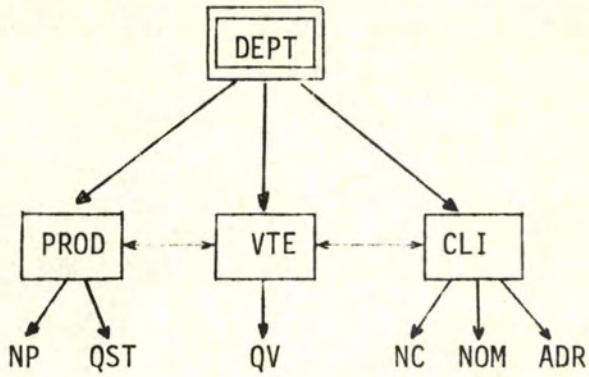
1.4.3. Exemple :

Il est sans doute utile d'illustrer concrètement ces notions.

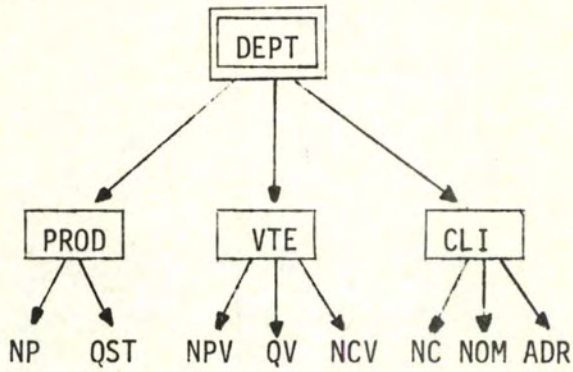
Prenons le cas d'un département d'entreprise qui s'occupe de la vente de produits finis aux divers clients.

- Les produits sont définis par un numéro et leur quantité en stock.
- Les ventes portent sur une quantité déterminée d'un produit et sont affectées à un (numéro de) client.
- Les clients sont répertoriés par leur numéro, leur nom et leur adresse.

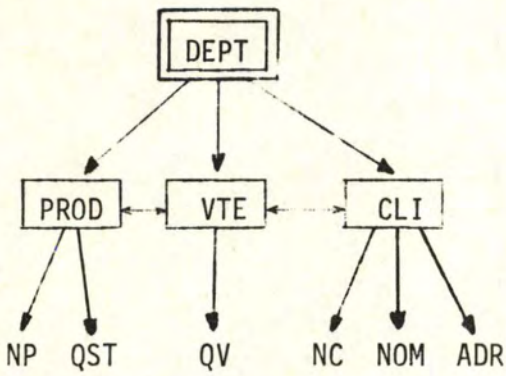
Un graphe de base sémantique de cette organisation peut être :



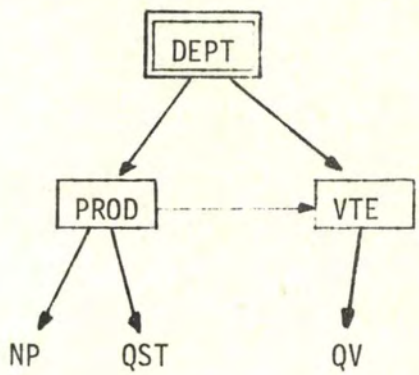
Le responsable de la BD peut fort bien avoir estimé que les fréquences d'utilisation des chemins d'accès PROD-VTE et VTE-CLI étaient trop faibles pour justifier le coût de leur implémentation, et il retient le graphe de base descripteur suivant :



Cependant, tous les utilisateurs n'ont pas la même vision de l'entreprise : le responsable de la gestion des stocks ne s'intéresse qu'à la quantité vendue globale et non ventilée sur les clients; on peut imaginer deux vues proposées :



Vue du directeur commercial



Vue du gestionnaire de stock

1.4.4. Correspondance entre modèles

Nous avons vu que la démarche de conception d'une BD procédait par niveaux d'analyse successifs partant des besoins à satisfaire en termes fonctionnels pour en arriver au niveau des choix et des contraintes techniques.

La structure de niveaux résultante peut aussi être perçue comme une architecture de SGBD : l'intégration des niveaux d'analyse entraîne alors la nécessité de pouvoir retrouver les éléments d'un modèle à partir des éléments d'un modèle de niveau inférieur. L'ensemble de ces règles de correspondance est appelée mapping.

D'autre part, les programmes d'application des utilisateurs s'appuient sur les éléments des modèles externes; il faut transformer ces programmes en des programmes qui travaillent sur le schéma interne. Une des solutions de ce problème est de créer un module de transformation, qui donnera la fonction de correspondance entre ces deux ensembles de programmes.

Reprenons l'exemple précédent et supposons qu'un utilisateur emploie dans un programme, une instruction d'accès de PROD à VTE : le module de transformation de programmes devra traduire cet accès par un accès séquentiel à VTE suivi du test d'égalité NPV OF VTE = NP OF PROD.

Là n'est pas la seule utilité d'un tel module : il ne rend pas nécessaire la réécriture d'un programme vis-à-vis des modifications, soit de la nature des informations que le programme n'utilise pas, soit de la structure des accès que ce programme utilise. C'est ce qu'on a coutume d'appeler l'indépendance des programmes par rapport aux données.

1.4.5. Choix d'un schéma interne

Le problème de choisir un schéma interne optimal, étant donnés un schéma conceptuel et la description d'un ensemble de programmes d'application, a été abordé par J.L. HAINAUT dans [2] et [13].

Brièvement, considérons un schéma conceptuel dont on connaît la description statistique et un schéma interne associé.

Imaginons un module de transformation qui permet de transcrire automatiquement un programme sur le schéma conceptuel en un programme sur le schéma interne, et aussi un module d'optimisation qui déterminerait automatiquement la forme optimale de ce dernier programme. (Les instructions du langage proposé sont traduites en un code intermédiaire qui mémorise la structure des accès d'un programme, comme nous le verrons plus loin)

Les éléments statistiques peuvent ainsi servir à évaluer le coût probable de chaque application et la fréquence d'utilisation de chaque relation, entre autres.

On peut encore envisager d'adopter une démarche itérative en choisissant une nouvelle structure d'accès, en fonction de ces quantifications et ce, jusqu'à atteindre un bon compromis dans le choix du schéma interne.

CHAPITRE 2 : Etude du langage de manipulation

2.1. Présentation

Le Langage de Description d'Algorithme (LDA ou ADL en anglais) est un langage destiné à commander des actions sur des collections de réalisations d'objets que l'on désigne préalablement.

La forme générale d'une instruction de ce langage est la suivante : "désignation d'une collection de réalisations d'un objet"
suivie d'une "séquence d'instructions".

Son interprétation sera : pour chaque réalisation de la collection faire
exécuter la séquence d'instructions
fin.

Plus concrètement, cette action sera exécutée comme suit :

1. obtenir le premier élément de la collection
2. exécuter complètement les instructions de la séquence pour cet élément
3. obtenir l'élément suivant de la collection
4. s'il existe, recommencer en 2
5. sinon, l'exécution de l'instruction est terminée.

Cette précision est importante dans le cas où la séquence d'instructions comporte une action de modification de la base de données telle que la collection de départ en soit affectée.

Du point de vue syntaxique, certaines instructions se présentent sous la forme simple < désignation > < séquence > (boucle implicite), alors que d'autres, telles que l'instruction d'accès, se présentent sous la forme explicite d'une boucle < désignation > < séquence > < fin > correspondant précisément à l'interprétation indiquée plus haut.

Ce langage est constitué de trois classes d'éléments :

- des instructions définissant une boucle de traitement, qui permettent de structurer un programme sous forme de blocs,
- des conditions qui permettent de désigner des collections et de contrôler l'ordre d'exécution d'une séquence d'instructions,
- des actions élémentaires qui ont un effet ponctuel et non plus itératif.

2.2. Généralités

2.2.1. Définitions

- Une séquence est une suite d'instructions (ou d'actions) qui devront être exécutées dans l'ordre pour chaque réalisation de la collection à laquelle elles sont directement associées.

  Au point de vue syntaxique, les différentes actions constituant une séquence seront séparées par des ";".

- La boucle de base d'une action (ou d'une séquence) est la boucle (implicite ou explicite) de désignation la plus imbriquée contenant l'action ou la séquence.

  Cette action sera exécutée pour chaque réalisation désignée par la boucle; la suite de ces réalisations sera appelée collection de base de cette action.

- Le corps d'une boucle est la séquence dont la boucle de base est cette boucle.

2.2.2. Conventions syntaxiques

Nous décrirons la syntaxe des formes du langage en usant du métalangage suivant :

- nous noterons en lettres majuscules les éléments faisant partie du langage (les terminaux),
- les termes entre < et > désignent des non-terminaux : ce sont soit des identificateurs, soit des expressions à développer,
- un terme souligné est obligatoire, qu'il soit terminal ou non,
- une liste de termes superposés entre accolades impose de choisir un et un seul terme parmi eux,
- $[< M >]_i^j$ signifie que le non-terminal < M > doit apparaître en un nombre d'exemplaires compris entre i et j; notons que < M > est mis pour $[< M >]_\phi^1$ puisque ce terme n'est pas souligné. Nous prenons cette convention pour alléger l'écriture de la syntaxe car certaines formes du langage proposent de nombreux paramètres facultatifs.

2.3. L'action BLOC

< PROGRAMME > ::= < bloc > [< bloc >]$_{0}^{\infty}$

< bloc > ::= < séquence LH > | < action bloc >

Un programme LDA est donc une suite de séquences de langage hôte (le COBOL en l'occurrence) ou d'actions "bloc" du type :

ENTER < nom racine > . < nom VD > < séquence > EXIT

où < séquence > est une liste d'instructions commandant des actions à exécuter dans le cadre d'une base de données dont la racine porte le nom < nom racine > (et qui est une collection de réalisations d'objet à un seul élément).

Les exemples suivants porteront sur la BD FNDP dont le diagramme est figuré page 4.

Exemple :

```
....
instructions COBOL
....
ENTER FNDP action 1; action 2;
          action 3
EXIT
....
instructions COBOL
....
```

Nous allons passer en revue les différentes sortes d'instructions tout en notant qu'une < séquence > peut contenir des actions < bloc > ou des < séquences LH >.

2.4. L'action d'ACCES

Syntaxe

Forme courte :

< accès > ::= [< relation > : < quantif. A> < nom O >.< nom VD > < condition > < séquence >]

Forme étendue :

REACH VIA < relation > : < quantif. A > < nom O >. < nom VD > < condition > < séquence > END

A partir de chaque élément de la collection de base de cette action,

- accéder par la relation de nom < relation > à certaines réalisations de l'objet de nom < nom O > ;
- ensuite, pour chacune de ces réalisations particulières, effectuer les actions prescrites dans < séquence > .

Le principe de base du LDA s'interprète donc comme suit : par un accès "filtré", on constitue une collection de réalisations de l'objet < nom O >, et pour chaque élément de cette collection, on effectue les actions décrites dans < séquence >.

Ce principe s'applique de manière récursive car < séquence > peut à nouveau contenir des instructions d'accès.

Exemples :

...[professeur : ALL PERSONNE (C1) action 1; action 2]

Cette instruction d'accès commande l'accès à tous les professeurs d'une faculté (de la collection de base) qui vérifient la condition C1 et pour chacun de ceux-ci, elle commande les actions 1 et 2.

...[professeur : ALL PERSONNE action 1]

Dans ce cas, la condition est absente et l'utilisateur se propose d'effectuer un traitement sur tous les professeurs d'une faculté; il peut aussi n'examiner qu'une partie des réalisations cibles :

...[professeur : 2# - 5# PERSONNE action 1]

Dans cette instruction, l'action 1 ne traitera que les 2°, 3°, 4° et 5° professeurs de la faculté relative à la collection de base.

...[: ALL FACULTE action 1]

Cette instruction commande l'accès (à partir de la racine) à toutes les facultés : il n'y a pas de clause < relation > puisque la relation FNDP - FACULTE est sans nom.

Nous avons évoqué plus haut le "filtrage" des réalisations de l'objet < nom O >: les exemples montrent que ce sont les termes <quantif. A >et< condition >qui indiquent la manière de sélectionner ces réalisations.

Le quantificateur d'accès est soit ALL, soit un ordinal; l'option par défaut est ALL.

- L'ordinal est la description d'une plage de valeurs spécifiant le rang des cibles à retenir, sous la forme m# - n# où m et n sont des nombres entiers positifs ou des variables de travail du type entier, tels que $m \leqslant n$ et $m, n \leqslant j_r$ si la relation r est de caractéristiques i_r - j_r, K_r - L_r; m# et n# peuvent s'écrire ## ("le dernier") et on admet la forme condensée n# pour n# - n# ("le n-ième").
- L'étude de la condition est reportée à un paragraphe ultérieur; dans les exemples, nous figurerons une condition par les symboles (C), (C1), ... et nous en donnerons toujours le sens.

  Exemple :

  [professeur : 2# - 5# PERSONNE (C) action]

  Si C signifie "dont l'adresse est NAMUR", cette instruction demande d'accéder du deuxième au cinquième professeur namurois relatifs à chaque réalisation de la collection de base (dans l'ordre physique de la base) et d'effectuer une action pour chacun de ceux-ci.

  Dans cette interprétation, le filtrage par la condition est réalisé avant le filtrage par l'ordinal : en d'autres termes, on comptera le nombre de réalisations cibles vérifiant la condition et on retiendra celles dont le rang appartient à la plage des ordinaux.

Si dans l'exemple, pour une faculté de la collection de base, il n'y a qu'un professeur de Namur, il ne sera pas traité par l'action suivante et si une autre faculté emploie 3 professeurs namurois, l'action traitera le 2° et le 3°.

- < Relation > peut spécifier une relation simple ou une composition de relations.

  On impose - de spécifier les objets intermédiaires afin de préciser de manière unique le chemin d'accès voulu (les relations sans nom peuvent être ambiguës) :

  < Relation > ::=[< nom relation >. < nom O cible >.] ϕ^{∞} < nom relation >

- de ne composer que des relations du type i - j, K - 1 ("one to many"), la première relation pouvant être de type quelconque.

La composition de relations n'est pas uniquement destinée à compacter l'écriture des boucles d'accès : elle permet d'ignorer les niveaux intermédiaires dans le contrôle imposé par le quantificateur.

Exemple :

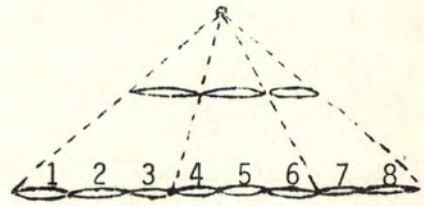
Supposons que l'on veuille subsidier les facultés en fonction des cours qu'elles organisent; convenons que ces subsides sont proportionnels au nombre global de cours.

Si l'on veut ne favoriser les facultés qu'à partir de l'organisation d'un troisième cours, on écrira FACULTE [. SECTION. : 3# - ## COURS action]

Schématisons la situation d'une faculté,

de ses sections,

de ses cours.



Cette faculté sera subsidiée sur l'organisation des cours 3, 4, 5, 6, 7, 8.

Si nous ne disposions pas de cette possibilité, nous devrions écrire

FACULTE [: ALL SECTION [: 3# - ## COURS action]

Dans ce cas, les subsides porteraient sur les cours 3 et 6 mais telle n'était pas notre intention.

Nous verrons par la suite que le compilateur génère les accès intermédiaires dans un accès par composition, en leur donnant le quantificateur ALL : cette forme générée diffère d'une suite normale de boucles d'accès imbriquées par le mécanisme de comptage qui est appliqué au plus haut niveau et non pas au dernier niveau intermédiaire.

Ainsi dans l'exemple, le compilateur décomptera les cours directement à partir d'une faculté et non pas à partir des sections de cette faculté.

- Notion de variable de désignation

Dans l'instruction d'accès, de même que dans la forme < bloc >, le terme < nom VD > qui suit < nom O > ou < nom racine > représente le nom d'une variable de désignation.
Une variable de désignation (VD) peut contenir la référence de la réalisation courante de l'objet < nom O > ou < nom racine >; elle permet d'utiliser cette référence :
- dans une condition où elle pourra par exemple faire l'objet de comparaisons,
- dans la liste d'actions qui la suit, à quelque niveau d'imbrication que ce soit.

Cette variable est attachée à la réalisation courante d'une boucle : c'est pourquoi nous parlerons de la "boucle < VD >" ou encore du "niveau < VD >".

De plus, cette variable est inconnue en dehors de l'instruction dans laquelle elle a été définie; elle a donc pour portée la boucle qui l'a définie.

- Notion de variable de travail

Une variable de travail (VT) permet de stocker la référence d'un élément d'une collection (de réalisations d'objet).

Si la variable est simple, chaque référence que l'on y range écrase la précédente. Sinon, les références sont empilées dans la variable dite "à empilement".

La portée d'une VT est le programme tout entier.

Des primitives de manipulation de VT sont exposées au paragraphe des actions élémentaires.

Dans le cadre limité de ce travail, seules les variables de travail simples ont été implémentées.

2.5. L'action sur une variable

Syntaxe

< Action V > ::= FOR < nom V > . < nom VD > < condition > < action >

Sémantique

Pour la réalisation référencée par chaque élément de la variable de nom < nom V >, qui vérifie la condition éventuelle < condition >, effectuer l'action du type < action > ; < nom V > est le nom d'une variable relative à un objet complexe; si l'on veut effectuer une action sur un objet élémentaire, il faut employer la forme <action OE>. Il faut remarquer que l'élément visé de la variable est unique dans le cas d'une VD ou d'une VT simple et qu'ici aussi, on peut référencer la réalisation courante de <nom V> par une VD à la condition évidente que < nom V >représente une variable de travail.

L'instruction <action V >vérifie le principe du LDA tout en imposant à la liste d'actions d'être limitée à un seul élément. La raison de cette limitation est d'ordre syntaxique. En effet, une deuxième instruction relative à la variable pourrait tout aussi bien se rattacher à la collection de base dont dépend < action V >.

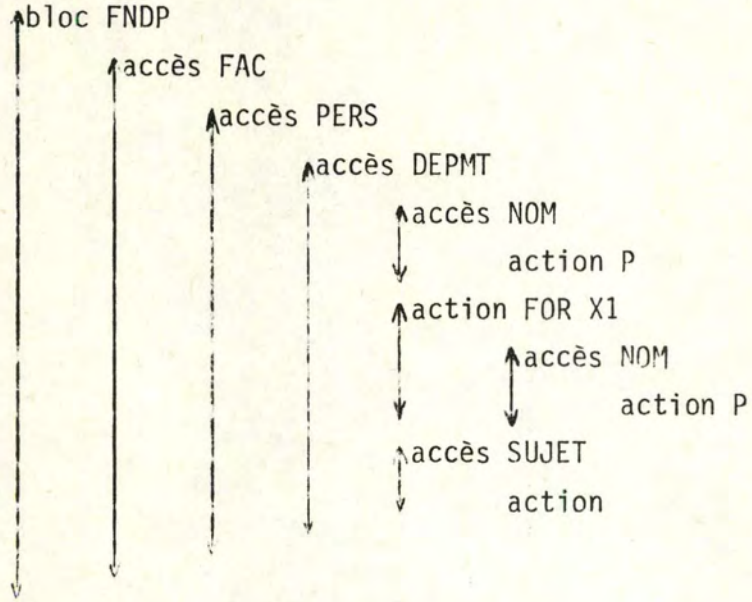
Exemple :

```
ENTER FNDP
      REACH : ALL FACULTE(C1)
            REACH VIA professeur : ALL PERSONNE.X1
                  REACH VIA directeur : ALL DEPARTEMENT [ :NOM P] ;
                                                        FOR X1 [: NOM P ] ;
                                                        [ :SUJET-RECH action ]
                  END
            END
      END
EXIT.
```

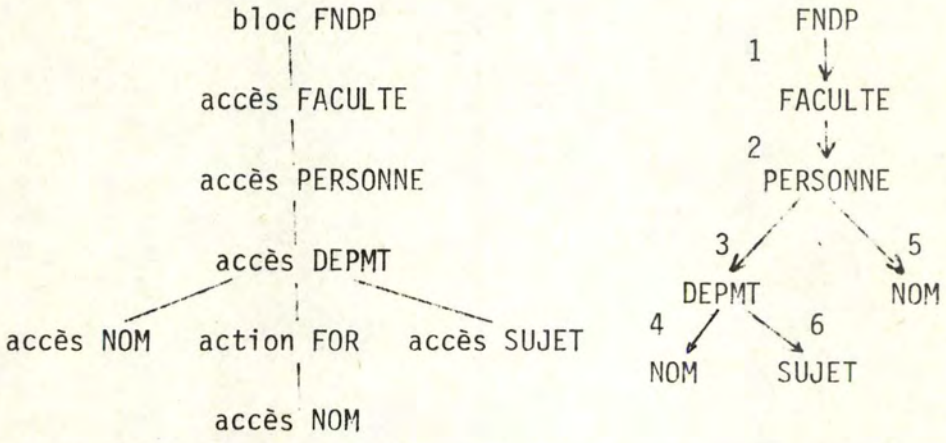
A partir de la base FNDP, on accède à toutes les facultés vérifiant C1; pour chacune d'elles, on accède à chacun de ses professeurs désormais référencé par la VD X1 et pour lequel on demande d'accéder à tous les départements dont il est directeur et d'imprimer le nom de chaque département (action P). Au lieu de poursuivre le chemin d'accès à partir du niveau où l'on est arrivé (ici, l'objet NOM relié à l'objet DEPARTEMENT), le programme commande de repartir de la réalisation courante de la personne (référencée par X1) afin d'en imprimer le nom; l'accès suivant aux sujets de recherche se réfère à un département et non plus à une personne.

Cet exemple montre l'utilité d'employer des variables, qui permettent de rompre l'arborescence imposée par les accès de la structure MSI.

Le schéma d'imbrication des instructions est en effet :



Nous pouvons comparer l'arborescence "de programme" représentant cette structure et l'arborescence des accès réalisés :



La numérotation des flèches indique l'ordre des accès.
On verra plus loin qu'un des soucis du code intermédiaire sera de réunir ces deux arborescences en une seule structure.
Si on veut commander plusieurs actions à partir d'une variable, on utilisera la forme < accès fictif V >.

2.6. L'action d'accès fictif à une variable

Syntaxe

< accès f.V >::= FOR< VD racine >[: <quantif. A > < nom V >< condition >
< séquence >]

Sémantique

Cette instruction donne la possibilité d'accéder à certaines réalisations d'un objet complexe, directement à partir de la racine référencée par< VD racine >; ce paramètre peut être omis si la boucle de base de cette action est une action BLOC.

Dans tous les cas, l'objet référencé par< nom V > doit appartenir à la BD identifiée par la racine,désignée explicitement ou implicitement.

Dans cette instruction, on imagine qu'il existe une relation fictive sans nom de la racine à chaque variable déclarée. Une fois cette convention admise, cette action obéit aux mêmes règles que la boucle d'accès classique. Elle permet en particulier l'utilisation d'un ordinal en plus du regroupement d'actions; sa signification est donc :
pour chaque élément de la variable < nom V > vérifiant la < condition > éventuelle (qui détermine le rang acceptable de cet élément), exécuter les actions énumérées dans la liste < séquence >.

Ici aussi, on peut adjoindre une < VD > dans le cas où < nom V > représente une variable de travail.

2.7. L'action à partir d'un objet élémentaire

Syntaxe

< Action OE > ::= < nom OE > < condition OE > < action * >

Sémantique

Pour chaque réalisation de l'objet élémentaire < nom OE > qui vérifie la condition < condition OE > , effectuer l'action désignée par < action *>.
Les instructions permises pour < action *> seront limitées à l'accès, à la création et à la suppression de relation.
On limitera également les caractéristiques de < condition OE > aux comparaisons de valeurs (entiers, réels, strings).

Le fait qu'une seule action puisse être spécifiée dans cette action a été déterminé pour la même raison que pour l'instruction < action V > .

Exemple :

NOM = 'DUPONT' [: ALL PERSONNE < action >].

L'effet de cette instruction est de se positionner dans la BD sur l'objet élémentaire NOM; pour toutes les réalisations dont la valeur est 'DUPONT', on commande d'accéder aux réalisations de PERSONNE qui leur sont liées, et d'effectuer une action pour chacune de celles-ci. Ceci revient à effectuer < action > pour toutes les personnes de nom 'DUPONT'.

2.8. L'instruction GENERATE

Syntaxe

GENERATE < nom O > . < VD > < séquence > END

Sémantique

Cette instruction a pour effet de définir une variable de désignation où sera rangée la référence à une réalisation de l'objet complexe ou élémentaire < nom O > .

La portée de cette variable correspond au bloc GENERATE ... END.

La VD déclarée par une instruction GENERATE sera vide jusqu'au moment où elle sera réutilisée sous la forme < cible > . < nom VD > , vue dans les actions d'accès (< cible > étant soit <nom O >, soit < nom VT >); elle contiendra la référence de la réalisation courante de la cible immédiatement après l'accès à cette réalisation.

Il faut évidemment que le type de l'objet associé à la déclaration (dans l'instruction GENERATE) soit le même que le type de l'objet associé à l'affectation (dans les actions d'accès).

Dans le cas où la variable < nom VD > n'a pas été déclarée par l'emploi d'un GENERATE préalable, sa déclaration et son affectation sont simultanées.

Notons que cette double forme de déclaration provient d'une économie d'écriture : la forme [: < nom O1 >.< VD1 >...] où <VD1 > est une variable non déclarée aurait pu s'écrire GENERATE < nom O1 >.<VD1 >FOR < VDØ >[:<nom O1 >.<VD1 >...] END où<VDØ > est le niveau de base de la boucle < VD1 >.

C'est cette dernière option qui a été prise pour la compilation : le compilateur génère automatiquement des instructions GENERATE pour les variables non déclarées ainsi que pour les actions qui ne mentionnent pas de variable d'affectation après la cible.

2.9. Les actions de modification de la base de données.

Ces actions ne sont pas reprises dans le cadre de l'implémentation : aussi nous bornerons-nous à les citer et à en donner une signification peu détaillée.

- L'action d'addition de réalisations.

Cette action permet d'ajouter une réalisation à l'objet complexe dont on cite le nom, à partir des renseignements fournis dans une condition.

- L'action de suppression de réalisations.

Cette action provoque la suppression de chaque élément de la collection (de réalisations d'un objet complexe) de base et soit une nouvelle itération prématurée, soit une fin prématurée de la boucle de base.

Cette double action est évidemment destinée à empêcher l'utilisateur de poursuivre un traitement sur des réalisations qui n'existent plus.

- L'action de création d'occurrences de relation.

Cette action a pour effet de créer des occurrences de la relation citée entre chaque élément de la collection de base et chaque élément d'un ensemble décrit par sa désignation.

- L'action de suppression d'occurrences de relation.

Cette action commande de supprimer les occurrences de la relation citée entre chaque élément de la collection de base et un élément d'un ensemble désigné.

- L'action de modification d'occurrences de relation.

L'effet de cette action est de remplacer les occurrences d'une relation, qui concernent chaque élément de la collection de base et les éléments d'un premier ensemble désigné, par des occurrences de cette même relation entre chaque élément de la même collection et les éléments d'un second ensemble désigné.

2.10. Les instructions de contrôle d'itération

Les instructions d'accès vues précédemment constituent un moyen simple d'obtenir toutes les réalisations d'un objet, ou une partie, vérifiant ou non certaines conditions et accessibles au moyen d'une relation d'accès et d'effectuer TOUTES les actions présentes entre les symboles extrêmes, pour CHACUNE de ces réalisations.

Il se peut cependant que l'on ne désire pas effectuer la totalité des actions pour chaque réalisation mais parfois passer à la réalisation suivante en abandonnant la fin des actions.

Il se peut encore que l'on veuille arrêter prématurément la ou les boucles dans lesquelles on se trouve.

Ces cas sont prévus par les ordres suivants :

2.10.1. Itération prématurée.

NEXT < nom VD >

Cet ordre commande l'arrêt du traitement de la réalisation référencée par la VD < nom VD > puis l'itération forcée de la boucle < nom VD >.
L'absence de < nom VD > signifie que l'on spécifie la boucle de base. Intuitivement, cet ordre correspond à un branchement au symbole de fin de la boucle relative à la VD.

2.10.2. Fin prématurée

EXIT < nom VD >

Cet ordre demande l'arrêt du traitement de la réalisation référencée par la VD citée puis la sortie de la boucle < nom VD > (ou de la boucle de base en cas d'absence de < nom VD >).
Intuitivement, cet ordre est assimilable à un branchement à la première action qui suit le symbole de fin de la boucle < nom VD > ou de la boucle de base.
Cette interprétation doit être précisée dans le cas d'un GENERATE.

```
GENERATE FACULTE.F1
        ....
        REACH FACULTE.F1
              ....
              EXIT F1
              ....
        END
        ...
END
```

L'ordre EXIT provoque la sortie de la boucle d'accès et non du contexte GENERATE.
Dans ce cas, le compilateur affectera une seconde variable à la réalisation courante de FACULTE afin de donner un nom différent aux 2 boucles.

2.11. Les actions élémentaires

2.11.1. Imprimer : PRINT

Cette action commande la visualisation de la valeur d'une réalisation d'un objet élémentaire ou la visualisation des valeurs des réalisations des OE directement accessibles à partir d'une réalisation d'un objet complexe.

Voici quelques primitives de manipulation de variables de travail.

2.11.2. → < nom VT >

La sémantique en a été vue dans le paragraphe "Notion de variable de travail". Rappelons qu'il y a deux types de VT : les VT simples et les VT à empilement. Cet ordre commande le rangement dans la VT < nom VT > de la référence de chaque élément de la collection de base.

2.11.3. CLEAR < nom VT >

Effacer le contenu d'une VT à empilement.

2.11.4. COMP < nom VT >

Compacter une VT en éliminant les duplications dans les références.

Exemple

On veut imprimer le nom de tous les professeurs des facultés.

```
ENTER FNDP [ : ALL FACULTE [ professeur : ALL PERSONNE → V1 ] ] ;
          COMP V1; FOR V1 [ : NOM P ] EXIT
```

Pour ce faire, on accède à partir de la racine à toutes les facultés; pour chacune d'elles, on accède à tous leurs professeurs, dont on empile les références dans V1.

Si on admet que plusieurs facultés peuvent employer la même personne, on élimine les références identiques dans V1. Enfin, pour chaque personne référencée par V1, on accède au nom et on l'imprime.

2.12. La condition

D'une manière générale, une réalisation soumise à une condition devra vérifier une expression booléenne de critères, écrite à l'aide des opérateurs &(ET), /(OU) et ¬(NON). L'opérateur ¬ est prioritaire sur l'opérateur &, lui-même prioritaire sur l'opérateur /.

Les critères sont de deux sortes :

- les critères d'appartenance qui imposent à la réalisation d'appartenir (ou non) à une collection désignée;
- les critères de relation qui imposent à la réalisation de participer à des occurrences de relation.

2.12.1. Les critères d'appartenance (CA)

Syntaxe : (= < désignation >) ou ¬ (= < désignation >)

Sémantique : les réalisations de l'objet, sur lequel porte le critère, sont retenues si elles appartiennent (ou non, si le critère est précédé de ¬) à la collection < désignation > .

A. L'objet visé peut être élémentaire

- < désignation > sera la description d'un domaine de valeurs compatibles que peut prendre cet OE : plage de valeurs, liste de valeurs, ...

  NO(=Ø-2Ø) désigne les réalisations de NO dont la valeur est comprise entre Ø et 2Ø.

  NOM(='SMITH', 'CARTER') désigne les réalisations 'SMITH' et 'CARTER' de NOM.

  DUREE(=6Ø-$MAX) désigne les réalisations de DUREE dont la valeur est comprise entre 6Ø et la valeur de la variable $MAX.

  RUE(=V2) désigne parmi les réalisations de RUE celles qui appartiennent aussi à V2 (du moins référencées par la variable V2).

- < désign. > pourra être aussi la désignation de réalisations d'objets reliés

  NOM(= NOM(: C1)) désigne les réalisations de NOM qui appartiennent aussi à l'ensemble des réalisations de NOM reliées à une réalisation référencée par la VD C1 (ceci est un critère de relation).

B. De tels critères peuvent aussi porter sur des objets complexes.

PERSONNE ¬(= FOR X1 [professeur : ALL PERSONNE])

Dans cet exemple, l'expression FOR X1 [professeur : ALL PERSONNE] désigne l'ensemble des professeurs de la faculté X1 et l'expression entière désigne les personnes qui n'appartiennent pas à cet ensemble.

Deux types de simplifications sont admis :

- les parenthèses peuvent être omises s'il n'y a pas d'ambiguïté et si elles ne jouent pas un rôle de priorité;
- si l'objet est élémentaire et de type alphanumérique à valeurs ordonnées, on écrira :

  (<21) ou <21 pour (= Ø-2Ø);

  (>2Ø) ou >2Ø pour (= 21-∞);

  (> 'DURAND') ou > 'DURAND';

  (<V1) ou < V1 si V1 est une VT simple qui contient une valeur compatible avec l'objet concerné; dans ce cas, NOM < V1 désignera les réalisations de NOM dont la valeur est inférieure à la valeur de la réalisation référencée par V1.

2.12.2. Les critères de relation (CR)

Syntaxe : (< relation > : < quantif. C > { < nom O > / < nom V > } . <nom VD ><condition>)

Sémantique : D'une manière générale, les réalisations de l'objet, sur lequel porte le critère, sont retenues si elles sont reliées à certaines réalisations d'un objet cité dans le critère et satisfaisant une éventuelle condition (qui peut être à nouveau un CR).

Ici encore, le filtrage s'effectue au moyen du quantificateur et de la condition citée dans le critère.

Exemples

FACULTE(professeur : ALL PERSONNE (C)) désigne les facultés dont tous les professeurs vérifient la condition C.

FACULTE(professeur : 1# PERSONNE (C)) désigne les facultés dont le premier professeur satisfait la condition C.

FACULTE(professeur : 1- PERSONNE (C)) désigne les facultés sont un professeur au moins satisfait (C).

FACULTE(professeur : 5+ PERSONNE) désigne les facultés ayant au plus 5 professeurs.

PERSONNE(: NOM = 'DUPONT') désigne les personnes dont le nom est DUPONT.

Dans ce dernier exemple, la relation concernée est sans nom et l'omission du quantificateur correspond à l'option par défaut 1-; la condition citée est un critère d'appartenance.

Le quantificateur du critère est un ordinal ou un cardinal.

Un cardinal est la description d'une plage de valeurs sous la forme m-n où
m et n ont les mêmes propriétés que dans l'ordinal.
m et n peuvent s'écrire ALL et on admet les formes condensées :

k pour k-k (k exactement),
k- pour k-ALL (k au moins),
k+ pour Ø-k (k au plus).

Notons que la convention implicite est 1- lorsque le quantificateur est omis.
Nous sommes à même d'interpréter plus précisément un CR dans les deux cas :

A. Le quantificateur est un cardinal

Formalisons un tel CR par l'expression $A(r_{AB}$: K B (C1))

A.1. B est un nom d'objet

Une réalisation a de l'objet A est retenue si, parmi les réalisations de l'objet B qui lui sont reliées par la relation r_{AB}, il s'en trouve p qui vérifient la condition C1 sur B, avec $p \in K$ (qui est une plage de valeurs).

Exemples :

PERSONNE(titulaire : 3+ COURS (C)) désigne les personnes qui sont titulaires d'au plus 3 cours vérifiant la condition C

PERSONNE (chercheur : 1- DEPARTEMENT = V1) désigne les personnes qui sont chercheurs dans au moins un département appartenant à la collection référencée par la variable V1, autrement dit, les chercheurs des départements "V1".

PERSONNE(: Ø PRENOM = 'WALTER') désigne les personnes dont aucun prénom n'a pour valeur 'WALTER'.

A.2. B est un nom de variable

a est retenue si parmi les réalisations référencées par B qui lui sont reliées par r_{AB}, on en trouve p qui vérifient la condition éventuelle.

Exemples :

- PERSONNE(adjoint : 1- V1 (: NO-ANNEE = '1977')) désigne les personnes qui sont adjoints à un cours au moins, qui soit référencé par V1 et qui soit donné en 1977, cette condition équivaut donc à :

PERSONNE(adjoint : 1- COURS ((= V1) & (: NO-ANNEE = '1977')))

- PERSONNE(: NOM = NOM (: F1)) désigne les personnes dont le nom appartient à l'ensemble des noms des facultés référencées par F1 c'est-à-dire les personnes ayant même nom que les facultés "F1".

B. Le quantificateur est un ordinal : A(r_{AB} : O B(C))

B.1. B est un nom d'objet

a est sélectionnée si pour tout p ∈ O, il existe une réalisation de B qui soit reliée à a par r_{AB}, de rang p et qui vérifie la condition éventuelle C. Si ## est mis pour n, la valeur effective de n sera le nombre de réalisations de B reliées à a; d'autre part, si ce nombre de réalisations cibles est inférieur à n, a sera néanmoins sélectionnée.

Exemple :

FACULTE(professeur : 2# - 4# PERSONNE(:ADRESSE(:VILLE = 'NAMUR')))

sélectionne les facultés dont les 2°, 3° et 4° professeurs habitent Namur. Supposons qu'une faculté a ait k professeurs.

- k<2 : a n'a pas de professeur ou n'en a qu'un ⇒ a est écartée
- k = 2 ⇒ a est retenue si son 2° professeur est namurois
- k = 3 ⇒ a est retenue si son 2° et son 3° professeurs sont namurois
- k⩾4 ⇒ a est retenue si ses 2°, 3° et 4° professeurs habitent Namur.

B.2. B est un nom de variable

a est sélectionnée si, pour tout p ∈ O, il existe une réalisation de l'objet assigné à B qui soit reliée à a par r_{AB}, dont le rang est p, qui soit référencée par la variable et qui de plus vérifie la condition éventuelle.

Exemple :

FACULTE(professeur : ## V1 (: PRENOM='NATACHA')) désigne les facultés dont le dernier professeur est référencé par V1 et a en outre pour prénom, Natacha.

Remarques

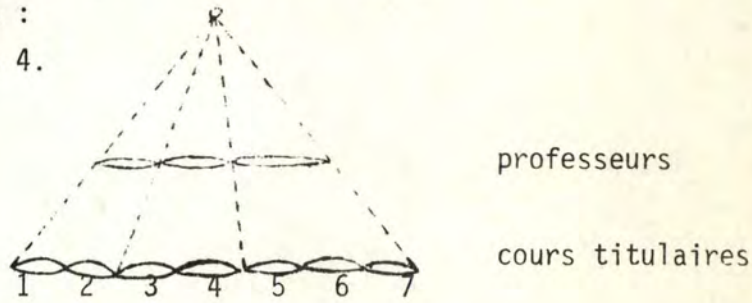
- L'ordinal du critère de relation a une signification inverse de celle de l'ordinal de l'accès : il faut traiter ici l'ordinal avant la condition et non plus après.
  En d'autres termes, le sous-ensemble de réalisations cibles qui va être examiné est désigné à priori par la plage d'ordinaux.
- Par contre, le cardinal du CR reprend le même type d'interprétation que l'ordinal de l'accès et l'on notera même que ALL est équivalent à 1#-##.
- <relation > peut être une relation simple ou une relation composée : cette dernière forme permet de négliger les niveaux d'accès intermédiaires en actionnant le filtre au seul niveau final, tout comme dans l'action d'accès.

Exemple :

On veut désigner les facultés dont le 4° cours, donné par un professeur titulaire, satisfait la condition C1; on écrira :

FACULTE(professeur. PERSONNE. titulaire : 4 # COURS (C1))

Prenons le cas d'une faculté a :
le cours impliqué est le cours 4.



Le compilateur va générer les critères de relation intermédiaires en leur assignant le quantificateur 1- mais en initialisant le compteur du filtre au plus haut niveau et non au dernier niveau intermédiaire, comme ce serait le cas dans l'expression :

FACULTE(professeur : 1- PERSONNE(titulaire : 4 # COURS (C1))) où le rang du cours serait décompté à partir du professeur titulaire et non de la faculté.

En outre, le compilateur doit ignorer de vérifier les critères qu'il a générés : dans l'exemple, cela mènerait à n'examiner aucun cours puisqu'aucun professeur n'est titulaire de 4 cours.

- Les réalisations de l'objet cible du CR, qui ont opéré la sélection, peuvent être référencées par une VD; la portée de cette variable est le CR tout entier qui l'a définie (de la parenthèse ouvrante à la parenthèse fermante).

- Condition associée à un objet élémentaire dans < action OE >.

Nous avons vu que l'action < action OE > demande de constituer à priori une collection de valeurs de l'objet élémentaire. A partir de chaque élément de cette collection, on commande un accès à une réalisation d'un objet complexe ou d'un OE composé (entre autres, du premier OE).
Ceci n'implique pas nécessairement que l'on doive emprunter une relation d'accès partant de l'objet élémentaire vers un autre objet : cet accès peut aussi être traduit par un accès séquentiel à l'objet visé avec vérification de la valeur de l'objet élémentaire.

Nous avons estimé que les possibilités du système cible, le DML de l'Institut, étaient suffisamment étendues pour les reprendre; aussi, nous admettrons les critères suivants :

< condition OE > ::= $\left\{ \begin{matrix} = \\ \neg = \end{matrix} \right.$ < liste valeurs > | < opérateur > < désignation OE >

< opérateur > ::= $\leq$ | $\geq$ | $\neg\leq$ | $\neg\geq$

< liste valeurs > ::= < désignation OE > [, < désignation OE >]$_{0}^{\infty}$

< désignation OE > ::= < entier > | < réel > | < chaîne > | < VT simple > | < VD >

2.13. La désignation

• De façon générale, la désignation comprend tout ce que nous avons vu jusqu'à présent : pour désigner une collection de réalisations d'un objet, nous avons décrit un chemin d'accès partant de la racine et aboutissant à l'objet désiré.

Ainsi l'expression ENTER FNDP [: 1# FACULTE [professeur : ALL PERSONNE]] EXIT désigne tous les professeurs de la première faculté de Namur.

Il peut arriver que l'on veuille restreindre cette collection : nous avons vu que l'on pouvait imposer des conditions à chaque noeud du chemin.

Si l'on veut désigner les professeurs de la faculté de droit, qui habitent Namur, on écrira ENTER FNDP [: FACULTE (: NOM='DROIT')

[professeur : ALL PERSONNE(: ADRESSE(: VILLE

= 'NAMUR'))]]

Comme tout langage de manipulation de données, le LDA traduit une analyse de traitement par les données : avant de commander un traitement sur un ensemble de données, il faut désigner cet ensemble. Un programme de manipulation bien construit veillera à ne pas accéder plusieurs fois aux mêmes données et dès lors l'utilisateur tracera un chemin d'accès en fonction de tous les traitements à faire.

Ainsi, s'il faut modifier certaines données d'une faculté et de quelques-uns de ses professeurs, on aura ENTER FNDP [: FACULTE (C1) < modifier données >

[professeur : PERSONNE (C2)

< modifier données > ...

Une expression générale du LDA prend dès lors la forme d'une arborescence d'accès, dont tous les noeuds peuvent supporter une condition et subir un traitement, comme l'a montré l'étude de l'action < action V >.

• D'autre part, la désignation doit être utilisée de manière restreinte dans les instructions de manipulation de relations ou dans un critère d'appartenance où figure le terme < désignation > .

Dans le cas d'un CA, comme PERSONNE¬(= FOR X1 [: ALL PERSONNE]), l'utilisateur a déjà désigné le premier niveau de comparaison (ici, PERSONNE) et il lui faut décrire le second niveau par un chemin d'accès y menant; ce chemin part d'une collection, que nous appellerons collection de départ (la faculté 'X1'). et se poursuit par accès successifs (ici, un accès à PERSONNE).

A. Désignation de la collection de départ

L'étude des actions du LDA a montré que l'on pouvait se positionner à un niveau par une des trois formes : -ENTER < nom racine >

-< nom OE > { < condition OE > / < condition > }

-FOR < nom V > < condition >

Nous transposerons ces formes dans le cadre d'une désignation restreinte en y apportant le changement suivant, qui traduit un souci de compacter l'écriture des conditions : le mot-clé ENTER sera interdit.

B. Désignation par accès

La désignation de départ sera suivie d'une série d'actions d'accès imbriquées ou d'actions à partir d'une variable; toute autre action est interdite.

Exemples :

PERSONNE(: NOM=NOM(:F1)) désigne les réalisations de PERSONNE dont le nom appartient à l'ensemble des noms reliés à la faculté référencée par la variable F1.
En clair, on recherche les personnes qui ont même nom que les facultés F1.

L'exemple suivant porte sur la structure décrite p. 35.

COMMANDE (=FOR XØ [: ALL CLIENT.X1(: NOM='DUPONT')
FOR XØ [: ALL COMMANDE (: NUMCLI=NUMCLI(: X1))]])*

L'expression FOR XØ [: ALL CLIENT.X1(: NOM='DUPONT') désigne tous les clients de nom Dupont (la variable X1 contiendra la référence de chacun de ces clients) tandis que l'expression FOR XØ [: ALL COMMANDE(: NUMCLI=NUMCLI(: X1))] désigne l'ensemble des commandes portant le même numéro de client que les clients référencés par la variable X1. L'expression entière désigne donc toutes les commandes des clients Dupont.

Cette forme équivaut à COMMANDE(: NUMCLI=NUMCLI(:1-CLIENT (: NOM='DUPONT'))).

Si l'on veut toutes les premières commandes des clients Dupont, on doit écrire :

COMMANDE(= FOR XØ [: ALL CLIENT.X1(: NOM='DUPONT')
FOR XØ [: 1# COMMANDE(: NUMCLI=NUMCLI(:X1))]])

---

* On suppose que XØ désigne la racine BASE.

2.14. L'instruction conditionnelle

Syntaxe :

IF < condition IF > THEN <séquence > ELSE < séquence > END

Sémantique

1° forme : IF (C) THEN action 1 END; action 2
si (C) est vraie, il faut exécuter action 1, puis action 2
sinon, passer directement à action 2.

2° forme : IF (C) THEN action 1 ELSE action 2 END; action 3
si (C) est vraie, il faut exécuter action 1 puis passer à action 3
sinon, passer à action 2 et exécuter action 3.

Toute condition LDA est relative à une réalisation d'objet. Dans le cas particulier de la condition de sélection d'une instruction d'accès, par exemple, la réalisation est celle qui vient d'être acquise par la boucle d'accès; il est donc inutile de la désigner explicitement.

Par contre, dans l'instruction conditionnelle, on désire étendre les possibilités en affectant la condition à toute réalisation courante à cet endroit, et non plus seulement à celle de la collection de base. Il peut donc être nécessaire de spécifier la réalisation concernée, ce qui se fera le plus simplement au moyen d'une variable de désignation :

```
IF (X1(C1)) THEN ... END
```

L'absence d'une telle VD indique donc que la condition porte sur la réalisation courante de la collection de base, comme dans le cas d'une condition dans un accès.

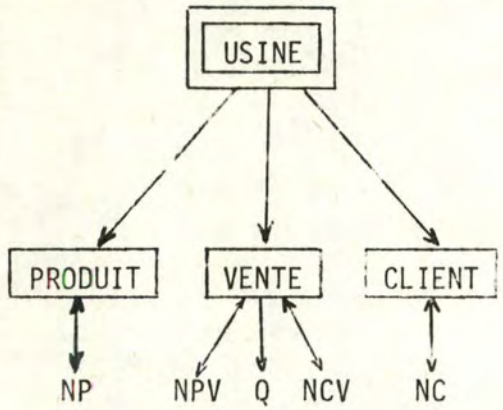
Une < condition IF > peut contenir des conditions relatives à plusieurs réalisations distinctes, comme dans l'exemple ci-dessous où BØ est une référence de la racine FNDP et P1, une réalisation de PERSONNE :

IF((BØ(: FACULTE((C1)&(C2))))&(P1(titulaire : COURS((C3)/(C4)))))... signifie : s'il existe une faculté vérifiant C1 et C2 et un cours dont P1 est titulaire, satisfaisant à C3 ou à C4, ...

2.15. Exemples

Exemple 1

Cette première série d'exemples s'applique à la base de données suivante :



NP, NPV : numéro de produit
NC, NCV : numéro de client
Q : quantité

1. Accéder aux ventes du client numéro 312.
   ENTER USINE [: ALL VENTE(: NCV=312)] EXIT

2. Accéder aux ventes des clients vérifiant la condition C1.
   ENTER USINE [: ALL VENTE(: NCV=NC(: CLIENT(C1)))] EXIT

3. Accéder aux clients n'ayant aucune vente.
   ENTER USINE [: CLIENT(: NC ¬=NCV(: 1- VENTE))] EXIT

4. Accéder aux clients ayant au moins une vente concernant le produit (C2).
   ENTER USINE [: CLIENT(: NC=NCV(: 1- VENTE(: NPV=NP(: PRODUIT(C2)))))] EXIT

5. Accéder aux clients n'ayant pas acheté plus de 15 produits (C3).

```
ENTER USINE.XØ
      [ : CLIENT.X1(: NC ¬= FOR XØ[ : 16# VENTE((: NCV=NC(: X1))
                                            &(: NPV=NP(: PRODUIT(C3))))
                                            [ : ALL NCV] )] EXIT
```

Cette formulation nécessite quelques explications : on recherche les clients n'appartenant pas à l'ensemble des clients dont on peut trouver la 16° vente du produit C3.

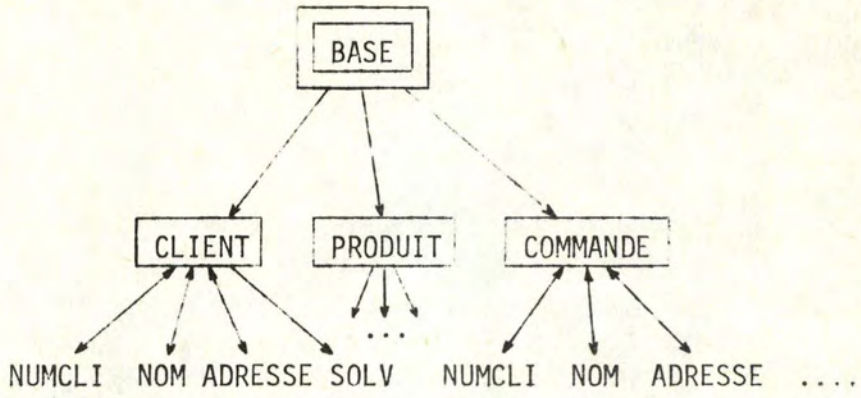
Le fait que cette expression soit longue et peu "naturelle" n'est pas imputable au LDA mais bien à l'absence d'une relation d'accès entre CLIENT et PRODUIT, auquel cas on écrirait plus simplement :

```
[ : CLIENT(: 15+ PRODUIT(C3))]
```

D'une manière générale, on peut dire que la complexité des expressions de désignation va de pair avec la pauvreté de la structure choisie du MSI.

Exemple 2

Cet exemple traite de la BD suivante :



Il faut mettre à jour les enregistrements CLIENT en fonction des bons de commande. La convention est que les bons de commande sont exacts; si les caractéristiques communes des clients et des commandes sont différentes, il faut donc corriger les premiers à partir des derniers.

Pour chaque commande C1 (1) si son numéro est le numéro d'un client,
- atteindre ce client et le référencer par CL1,
- si son adresse diffère de celle de la commande,
  la modifier;sinon passer à (2)

sinon créer dans CL1 un client sur base de la commande

(2) si ce client n'est plus solvable, éditer un message et passer à la commande suivante

sinon, ...

On adoptera la syntaxe étendue pour rédiger ce programme.

```
ENTER BASE.BØ
      REACH : ALL COMMANDE.C1
            GENERATE CLIENT.CL1
                IF(BØ(:1- CLIENT(: NUMCLI=NUMCLI(:C1))))
                  THEN FOR BØ REACH: CLIENT.CL1(: NUMCLI=NUMCLI(:C1)) END;
                       IF(CL1(: ADRESSE ¬=ADRESSE(: C1)))
                         THEN
                            < modifier adresse client CL1 >
                       END
                  ELSE A CLIENT.CL1 ((: ADRESSE=ADRESSE(: C1))& ...)
                END;
                IF(CL1(: SOLV='-') THEN < édition message >;
                                        NEXT C1
                END;
                ...
            END
      END
EXIT
```

CHAPITRE 3 : Le DML de l'Institut

3.1. Objectifs

Dès l'abord, ce langage a été conçu comme un intermédiaire entre ADL et un langage de manipulation de données d'un système de gestion de fichiers ou de banques de données (ici SESAM du constructeur SIEMENS).

En complément, ce système devait répondre au double objectif suivant :

- être un langage de haut niveau afin de pouvoir y rédiger facilement des programmes de signification claire;
- permettre au programmeur l'écriture de programmes qui optimisent la place en mémoire centrale et le nombre d'accès à la base de données, par une gestion tour à tour implicite et explicite de la mémoire et des demandes d'accès.

3.2. Principes du langage

Parmi les langages de bases de données, on peut distinguer deux classes :

- Certains langages permettent d'interroger une BD de manière ponctuelle : ils fournissent des primitives d'accès ponctuel du genre "accéder au record qui a telle clé d'accès", "accéder au record suivant"...
- D'autres langages le font de manière globale en fournissant des primitives du type "pour chaque record vérifiant certaines propriétés, effectuer telle liste d'opérations".

Par conséquent, dans les premiers langages, ces dernières primitives doivent être construites sous forme d'une boucle utilisant plusieurs primitives d'accès ponctuel, dont le déroulement est à contrôler au moyen d'un indicateur.

Le DML fait partie des langages du second type, tout comme ADL; ces langages considèrent un accès ponctuel comme un cas particulier de la boucle d'accès.

Signalons encore que les macros du DML sont insérées dans un contexte de COBOL.

3.3. Les différents ordres du DML

3.3.1. Définition d'un contexte associé à une BD

Un programme peut travailler sur une ou plusieurs BD : c'est pourquoi il faut déclarer par une commande spécifique le nom de la racine du graphe décrivant la BD à laquelle on s'intéresse.

Si nous désirons travailler sur la base de nom FNDP, nous l'indiquons par @OPEN FNDP.

Nous pouvons alors rédiger des instructions relatives à FNDP.

Lorsque nous désirons terminer le travail sur FNDP, nous écrivons @CLOSE.

Le bloc "@OPEN ... @CLOSE" définit un contexte dans lequel il est permis de travailler sur la base indiquée.

Il peut s'avérer utile d'associer une étiquette à la réalisation courante de l'objet racine; par extension, cette étiquette identifiera le bloc correspondant : par exemple, @OPEN FNDP = FAC précise que la base FNDP portera le nom particulier FAC dans ce bloc. Si un programme contient plusieurs blocs, il faut veiller à respecter les règles usuelles de disposition relative de différents blocs, tout comme en ADL.

3.3.2. Accès aux réalisations d'un objet complexe

A. Accès à toutes les réalisations d'un OC

L'accès se fera en empruntant la relation sans nom qui existe de la racine vers l'OC en question.

A.1. *Cet ordre est la première macro après l'OPEN correspondant*

Si PERSONNE est cet objet et < Séquence de traitement > sont les instructions à effectuer pour chaque réalisation obtenue, on écrira :

```
@OPEN FNDP=FAC.
      < Séquence 1 >
      @REACH PERSONNE=P1.
            < Séquence de traitement >
      @END.
      < Séquence 2 >
@CLOSE.
```

S'il n'existait aucune réalisation de PERSONNE au moment de l'exécution, le déroulement du programme se borne à l'exécution de < Séquence 1 > puis < séquence 2 >. Les règles relatives aux blocs OPEN-CLOSE sont aussi applicables aux blocs REACH-END.

En particulier, toute référence aux réalisations de PERSONNE est interdite dans < Séquence 1 > et dans < Séquence 2 >.

On peut aussi associer à l'ordre REACH, une étiquette qui pourra servir à référencer la réalisation courante de PERSONNE ici, dans le contexte de la boucle.

A.2. *Un tel ordre est séparé de l'ordre OPEN relatif, par plusieurs autres blocs.*

Pour éliminer les risques d'ambiguïté, il est possible d'indiquer dans la macro REACH à quelle racine on se réfère, au moyen du paramètre FROM.

Supposons que les bases FNDP et FOURNISSEUR possèdent toutes deux un objet PERSONNE .

```
@OPEN FNDP=FAC.
      @OPEN FOURNISSEUR.
            @REACH PERSONNE = P1 .
                  < Séquence 1 >
            @END .
            @REACH PERSONNE=P2 FROM FAC .
                  < Séquence 2 >
            @END .
      @CLOSE .
@CLOSE.
```

Sans le paramètre FROM FAC, la dernière boucle REACH parcourerait une deuxième fois les réalisations PERSONNE de FOURNISSEUR, alors que l'on veut examiner toutes les réalisations de PERSONNE de la base FNDP.

B. Accès aux réalisations accessibles à partir de certaines valeurs d'OE.

Il suffit alors de spécifier dans le paramètre IF les conditions imposées sur ces OE.

Exemple :
```
@REACH PERSONNE=P1 IF (NOM="GUILLAUME").
            < séquence >
      @END .
```

Cette boucle commande l'accès aux réalisations de PERSONNE dont le nom est Guillaume et pour chacune d'elles, l'exécution de < Séquence >.

Les conditions correspondent aux critères d'appartenance d'ADL, restreints aux OE.

C. Accès à des réalisations d'OC à partir d'autres réalisations d'OC

Utilisons par exemple la relation d'accès "professeur" de PERSONNE à FACULTE .

```
@REACH PERSONNE=P1 IF (DATE-ENTREE < 1970) .
       @REACH FACULTE=F1 FROM P1 VIA PROFESSEUR .
              < Séquence >
       @END .
@END.
```

L'ordre d'accès spécifie donc l'étiquette du REACH qui fournit les réalisations origines et le nom de la relation d'accès (si celle-ci porte un nom).

Il est permis d'omettre le paramètre FROM lorsque l'origine est fournie par l'ordre REACH précédent le plus proche.

3.3.3. Modification de la gestion automatique des boucles d'accès

Les ordres REACH et END fournissent le moyen d'obtenir toutes les réalisations d'un OC et d'effectuer toutes les opérations comprises entre ces ordres pour chacune de ces réalisations. Il se peut cependant que l'on veuille passer prématurément à la réalisation suivante, en abandonnant la fin des opérations pour la réalisation courante ou même arrêter le traitement d'une boucle.

L'ordre NEXT provoque l'accès à la réalisation suivante de l'ordre REACH spécifié par une étiquette à la suite du mot NEXT.

L'ordre EXIT provoque la fin anticipée d'une boucle identifiée par son étiquette. Si EXIT est employé sans étiquette (tout comme NEXT), le REACH concerné est le précédent le plus proche.

Exemple :

```
MOVE Ø TO COUNT.
@REACH FACULTE=F1.
       ADD 1 TO COUNT.
       IF COUNT > 2 GO TO SUITE.
       @NEXT.
       SUITE. < Séquence 1 >
@END.
```

Cette séquence commande l'accès aux facultés puis à partir de la troisième, l'exécution des instructions de < Séquence 1 >

3.3.4. Accès aux réalisations d'un OE.

La désignation des réalisations d'OE auxquelles donne accès une réalisation d'OC, se fera en conformité aux règles de qualification du COBOL.

Si FAC désigne une réalisation de FACULTE, on écrira RUE OF ADRESSE OF FAC. Préalablement, il faut indiquer la liste des OE reliés à l'OC (auquel on accède) et dont on désire obtenir les valeurs; cette liste est spécifiée dans le paramètre GET de l'ordre REACH.

Il existe un compteur de valeurs associé à chaque OE : ainsi, le nombre de valeurs de PRENOM associé à la réalisation identifiée par l'étiquette P1 s'indiquera par @COUNT PRENOM FROM P1.

Les valeurs de ces compteurs peuvent être consultées et mises à jour par le programmeur.

3.3.5. Gestion de la mémoire centrale

A. *Gestion automatique*

La mémoire centrale est gérée comme une pile :

- Lors d'un ordre REACH, une zone est réservée au sommet de la pile afin d'y accueillir toutes les données relatives à chaque réalisation fournie par cet ordre; ces données sont composées des valeurs d'OE demandées par le paramètre GET et de certaines données de gestion du système.
- Lors de la sortie d'un contexte par END ou par EXIT, la place réservée pour ce bloc est désallouée.

B. *Modification de la gestion automatique*

L'ordre ALLOCATE a pour effet de réserver dans la pile une zone où un ou plusieurs ordres REACH pourront ranger ultérieurement les données de réalisations d'un OC; s'il faut réserver de la place pour des valeurs d'OE, on utilisera le paramètre WITH (de la même façon que le paramètre GET de la macro REACH).

Cet ordre sera clôturé par un ordre END libérant la zone et délimitant un contexte de référence de la réalisation rangée.

Lorsqu'on désire accéder à des réalisations mais que l'on veut que les données en soient rangées non pas au sommet de la pile, mais dans une zone réservée par un ALLOCATE, on écrira @REACH PERSONNE=P2 IN P1, dans le contexte @ALLOCATE PERSONNE=P1 WITH ALL.

3.3.6. Les variables de travail

Une variable de travail (VT) peut contenir la référence à une réalisation d'OC. La portée de cette variable est constituée par le programme tout entier.

- L'ordre SAVE permet de sauver la référence de la réalisation courante associée à un REACH (ou à un ALLOCATE).

  On écrira par exemple @SAVE PERSONNE IN $45 dans le contexte de PERSONNE.

- L'ordre MOVE rend possible le rangement du contenu d'une VT dans une autre. On écrit @MOVE $64 TO $45.
- L'ordre TEST permet de vérifier si une VT ou une étiquette se réfère ou non à une réalisation : si oui, un registre spécial SYS-REG est positionné à 1.

  Cet ordre permet encore de comparer les réalisations référencées par deux variables, deux étiquettes ou une variable et une étiquette.
- Une autre utilisation de l'ordre REACH consiste à ranger dans la pile des données relatives à la réalisation référencée par une VT.

  On écrira par exemple @REACH PERSONNE $45=PERS3.

3.3.7. Les ordres de manipulation de données

Ces ordres ne sont pas exposés car ils ne seront pas utilisés dans l'implémentation de ce travail; cependant, nous voyons l'ordre de création d'une réalisation d'OC car il en sera fait mention dans l'exemple général suivant.

Cette fonction sera effectuée en trois étapes :

A. Réservation dans la pile d'une zone qui contiendra les valeurs des OE de la réalisation à créer.

   Cette réservation sera commandée par l'ordre PREPARE qui spécifiera les OE auxquels on désire affecter une valeur (de même que pour l'ordre ALLOCATE).

   La zone réservée sera désallouée par un ordre END, qui clôture le contexte de référence à la réalisation rangée (donc après l'exécution de l'ordre CREATE).

B. Rangement des valeurs d'OE dans cette zone au moyen d'instructions COBOL.

C. Rangement de la réalisation préparée dans la base par un ordre CREATE, qui mentionne les relations dont l'objet auquel on ajoute une réalisation, est origine; il est également permis d'y demander que la réalisation soit cible d'une relation, auquel cas on indiquera l'inverse de cette relation.

3.4. Conclusion et exemple

Les similitudes entre ordres du DML et du LDA sont apparues tout au long de cet exposé. Nous voudrions dès lors insister sur les caractéristiques originales du LDA.

- Il uniformise l'accès des OC et des OE .
- Il oppose une écriture compacte à la forme plus lisible du DML; nous pensons qu'un jugement en telle matière ne peut être dénué de subjectivité : aussi, nous bornerons-nous à citer le pour et le contre des deux syntaxes.

  La syntaxe compacte du LDA est peu explicite pour un observateur non averti, car elle permet d'exprimer des désignations assez complexes en peu de mots; il existe cependant des langages qui sont encore plus concis et qui jouissent d'une popularité certaine, comme APL.

  La syntaxe "naturelle" du DML est plus prolixe; dans certains cas, un programme DML sera plus compréhensible pour un non-initié tandis que dans des cas plus complexes, le programme peut être long et ceci peut empêcher le lecteur d'en saisir rapidement l'essentiel.

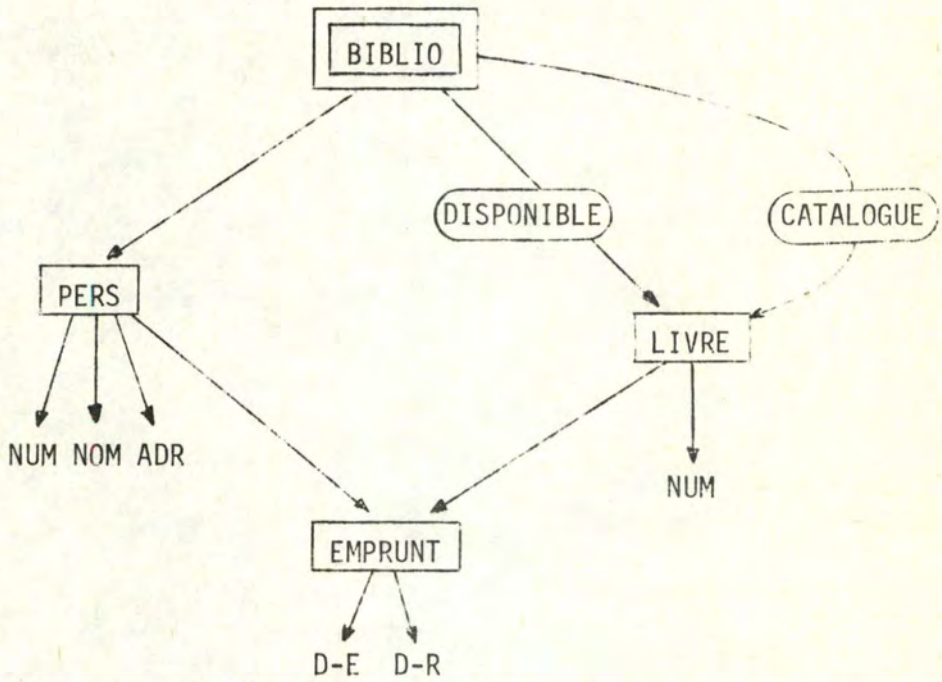
  Nous pensons que la forme étendue de l'action d'accès du LDA constitue un bon compromis entre ces deux optiques.
- En dehors de cette question de forme, ADL permet de formuler certaines requêtes assez simples dans un style comparable à celui de SEQUEL, les mots-clés exceptés, comme nous le verrons dans le chapitre suivant.
- ADL confère la possibilité de poser des questions complexes en demandant à une réalisation de participer à des occurrences de relations entre objets COMPLEXES.
- Il permet encore de contrôler la réponse à de telles conditions au moyen de l'instruction IF.
- Ces éléments font qu'ADL est plus indépendant du COBOL que le DML, de telle sorte que le premier donne les moyens de décrire presque complètement la structure d'un programme du point de vue de sa logique d'enchaînement des actions.

Pour illustrer ces avantages, considérons la gestion élémentaire d'une bibliothèque. Les livres sont répertoriés par volume, par leur titre et leur(s) auteur(s).

Les emprunts portent sur un volume et ont une date d'emprunt et une date de retour.

Les personnes sont identifiées par un numéro, leur nom et leur adresse.

Un diagramme du MSI peut être :



La procédure d'emprunt d'un livre donnera lieu aux opérations suivantes :

- Accéder au livre par son numéro .
- S'il est absent, c'est terminé ;
  sinon, . accéder à l'emprunteur par son numéro
  . si cette personne n'est pas enregistrée, la créer et aller à (1)
  sinon . accéder à son dernier emprunt
  . si la date de retour est égale à zéro, l'emprunt est refusé
  sinon, (1) créer l'emprunt.

Supposons encore que les emprunts soient classés en ordre inverse de leur ancienneté. Il y a plusieurs relations d'entrée aux livres; parmi elles, nous n'emploierons que la relation de nom DISPONIBLE qui accède aux livres disponibles pour un emprunt.

Le programme DML réalisant ce traitement sera :

```
@OPEN BIBLIO=B1.
  @ALLOCATE PERSONNE=P1 WITH ALL.
  @ALLOCATE LIVRE=L1.
     @REACH LIVRE=L2 FROM B1 VIA DISPONIBLE IF (NUM=N) IN L1.
     @END.
     @TEST L1.
        IF SYS-REG=Ø THEN DISPLAY "LIVRE NON DISPONIBLE" UPON TERMINAL
                          @EXIT P1.
     @REACH PERSONNE=P2 FROM B1 IF (NUM=X) IN P1.
     @END.
     @TEST P1.
        IF SYS-REG=1 GO TO PERS-EXIST.
     @PREPARE PERSONNE=P3 IN P1.
              MOVE X TO NUM OF P1
              MOVE Y TO NOM OF P1
              MOVE Z TO ADR OF P1.
              MOVE 1 TO @COUNT NUM FROM P1,
                        @COUNT NOM FROM P1,
                        @COUNT ADR FROM P1.
     @CREATE P1.
      GO TO CREAT-EMPR.
  PERS-EXIST.
              @ALLOCATE EMPRUNT=E1 WITH ALL.
              @REACH EMPRUNT=E2 FROM P1 IN E1.
                    @EXIT.
              @END.
              @TEST E1.
                 IF  SYS-REG=Ø OR D-R OF E1=Ø
                    THEN DISPLAY "EMPRUNT REFUSE" UPON TERMINAL
                         @EXIT P1.
  CREAT-EMPR.
              @PREPARE EMPRUNT=E3 IN E1.
                       MOVE Ø TO D-R OF E1.
                       MOVE TODAYS-DATE TO D-E OF E1.
                       MOVE 1 TO @COUNT D-R FROM E1,
                                 @COUNT D-E FROM E1.
              @CREATE E1, * : P1, * : L1.
              @END E1.
  @END L1.
  @END P1.
@CLOSE.
```

En voici la version d'ADL.

```
ENTER BIBLIO.B1
  IF(B1(DISPONIBLE : LIVRE (:NUM = N)))
    THEN GENERATE PERSONNE. P1
          IF(B1(:PERSONNE(:NUM=X))) THEN REACH PERSONNE. P1 (:NUM=X)
                                    ELSE ADD PERSONNE. P1 ((:NUM=X)
                                                          &(:NOM=Y)
                                                          &(:ADR=Z))
                                    END;
          IF(P1(:1#EMPRUNT(:D-R=Ø)))THEN
                                       < écrire "emprunt refusé" >
                                ELSE ADD EMPRUNT ((:D-R=Ø)
                                                  &(:D-E=TODAYS-DATE)
                                                  &(*:P1)
                                                  &(*:LIVRE(:NUM=X)))
                                END
          END
    ELSE
        < écrire "livre non disponible" >
    END
EXIT
```

CHAPITRE 4 : Comparaison des principaux DML.

4.1. Introduction

Le but de cette comparaison est d'esquisser les caractéristiques d'un certain nombre de langages de manipulation de données aux points de vue de la structure d'un programme, de sa lisibilité et de la manière de désigner une collection de réalisations.

Nous essayons donc essentiellement de nous mettre à la place d'un utilisateur peu spécialisé, mais nous verrons que certains langages réclament une spécialisation certaine.

La structure d'un programme de manipulation de données dépend d'une part du type de langage dans lequel il est écrit et d'autre part du type de modèle sur lequel il s'appuie.

4.1.1. Types de langages

- Dans un langage procédural, l'utilisateur est obligé d'accéder par programme à chaque donnée qu'il doit manipuler; c'est à l'utilisateur qu'incombe la charge de vérifier si l'accès s'est effectué avec succès et d'orienter le déroulement de son programme; ces tests seront exprimés dans un autre langage, du type classique, appelé langage hôte.
- Dans un langage non procédural, l'utilisateur n'a pas à se soucier de l'ordre dans lequel il manipule les éléments : il s'exprime en termes ensemblistes (ces langages sont parfois des langages de boucles) en laissant le soin au système qui interprête le langage, d'effectuer les accès nécessaires aux données.

4.1.2. Types de modèles

- Le modèle relationnel

L'approche relationnelle est basée sur la théorie mathématique des relations. Etant donnés n ensembles D_1 , D_2 , ... D_n(non nécessairement distincts), une relation est :

$R(D_1 , D_2 , \ldots , D_n) \subset \{(d_1 , d_2 , \ldots , d_n) \mid d_1 \in D_1$ et $d_2 \in D_2$ et ... et $d_n \in D_n\}$

$D_1 , D_2 , \ldots , D_n$ sont appelés les domaines de R et la valeur n est appelée le degré de R.

Pratiquement, on représente une relation par une table de valeurs dont chaque colonne correspond à un domaine et dont chaque ligne figure un n-uple de la relation.

Le fait qu'une relation soit un ensemble, a pour conséquences qu'il n'y a pas deux lignes identiques et que l'ordre des lignes est insignifiant; on peut encore imposer que l'ordre des colonnes soit insignifiant et que chaque valeur de domaine dans chaque n-uple soit une donnée indécomposable (comme un nombre ou une chaîne de caractères) si la relation est dite "normalisée".

Au point de vue de la manipulation, l'utilisateur a devant lui une série de relations (tables) qui sont associées par des domaines (colonnes) communs ou du moins compatibles.

Exemple : Un client est défini par l'ensemble des caractéristiques : numéro,
nom,
adresse.

Un produit est répertorié par son nom,
son type,
sa quantité en stock.

Une commande porte sur un numéro de client,
un nom de produit,
une quantité,
une date.

CLIENT	(NUMCLI, NOMCLI, ADRESSE)
PRODUIT	(NOMPRO, TYPE, QSTOCK)
COMMANDE	(NUMCLI, NOMPRO, QTE, DATE)

- Le modèle hiérarchique

Dans l'exemple présenté, on pourrait concevoir que les commandes soient regroupées et subordonnées à leur client respectif : l'utilisateur verrait un certain nombre d'articles, un par client, qui consistent en la réunion d'un article du type CLIENT et d'autant d'articles du type COMMANDE qu'il y a de commandes pour ce client. Cette réunion ou relation est en fait réalisée par juxtaposition physique ou par l'intermédiaire de pointeurs.

Conceptuellement, le diagramme d'une BD sera une structure arborescente de types d'articles,dont les arcs figureront les relations hiérarchiques.

Il importe de réaliser que tout article (non racine) doit avoir un "père" et un seul, ce qui peut paraître limitatif et qu'entre autres conséquences, l'utilisateur devra décrire le chemin hiérarchique nécessaire pour retrouver un article particulier.

- Le modèle de réseau

Ce modèle est en fait une généralisation du modèle hiérarchique : les noeuds d'un réseau représentent des types d'articles (e.g. CLIENT) et un noeud donné peut avoir un nombre quelconque de supérieurs immédiats et non plus un seul comme dans les hiérarchies.

Les articles subordonnés sont liés à un article supérieur : c'est pourquoi le langage de manipulation doit permettre à l'utilisateur de traverser les différents liens.

La figure suivante classe les langages étudiés en fonction de leur type et de la catégorie de modèle sur lequel ils travaillent.

MODELE / LANGAGE	HIERARCHIQUE	RESEAU	RELATIONNEL BINAIRE	RELATIONNEL N-AIRE
PROCEDURAL	IMS	IDS (CODASYL)		
NON PROCEDURAL	SOCRATE		DML INSTITUT ADL	SEQUEL

Le lecteur pourra paraître étonné de ne pas retrouver le modèle MSI (et donc les langages DML et ADL) sous la rubrique des réseaux : l'ambiguïté pourrait naître de la présentation graphique, sous forme de réseau de relations entre objets.

C'est essentiellement une différence de point de vue.

Le MSI est défini comme un modèle relationnel binaire : il se doit donc de présenter des relations "one to one" et "many to many" en plus des relations "one to many" proposées par un modèle en réseau; il doit encore être possible de déclarer qu'un objet est en relation avec lui-même, ce qui n'aurait pas de sens dans l'optique hiérarchique d'un réseau.

Signalons enfin qu'il est parfaitement possible de transcrire un diagramme usuel du MSI en un ensemble de tables binaires, dont chaque colonne contient les identifications de réalisations d'un objet.

4.2. Les langages des modèles relationnels

- Voici d'abord la table de correspondance entre le modèle relationnel et le MSI.

Base de données	- Racine
Relation A	- Objet complexe A relation (Racine, A) : Ø-∞, 1-1
Domaine B	- Objet élémentaire B
Appartenance du domaine B à la relation A	- Relation (A, B) : 1-1, Ø-1

- Il existe deux classes de langages définis à partir d'un modèle relationnel :
  (1) les langages de type algébrique ("relational algebra")
  (2) les langages de type prédicatif ("relational calculus").

Dans un langage algébrique, l'utilisateur doit spécifier les opérations d'algèbre relationnelle à exécuter pour produire le résultat escompté; une telle opération a pour opérande(s) une ou plusieurs relations et pour résultat, une nouvelle relation.

Les opérations fondamentales sont :

. la projection d'une relation sur un ou plusieurs domaines, qui consiste intuitivement à éliminer les autres colonnes de la table visée;

. la composition de deux relations sur un ou plusieurs domaines communs, qui revient au contraire, à fusionner deux tables.

Un programme de recherche dans un tel langage consistera en une suite d'opérations imbriquées; l'expression résultante sera certes concise mais généralement peu lisible pour un utilisateur peu versé en mathématiques.

Dans un langage prédicatif, il suffit de définir le résultat voulu en termes du calcul des prédicats, en laissant au système le soin de déterminer les opérations nécessaires; on passe ainsi à un type de langage non procédural.

Un tel langage a les caractéristiques suivantes :

. l'utilisateur définit des variables qui ont pour valeur une ligne d'une table "relation",

. il pose sa question en employant des prédicats et des quantificateurs existentiels et universels.

Par exemple, si nous voulons connaître la liste des numéros des clients qui ont passé toutes leurs commandes après une date donnée d, il faut poser une question du genre :

$\{NCLI.v \mid (v \in COM) \text{ et } \forall w(((w \in COM) \text{ et } (NCLI.v = NCLI.w)) \Rightarrow (DATE.w > d))\}$

Le principe est d'assigner la variable v à une ligne de commande; un algorithme correspondant de recherche pourrait être de vérifier si toutes les autres lignes de commande, assignées à la variable w, ont le même numéro de client que celui de la variable v : si oui, la variable v référence le numéro d'un client répondant à la question.

Cette expression est intéressante mais sa compréhension est encore réservée aux utilisateurs rompus aux subtilités de la logique des prédicats.

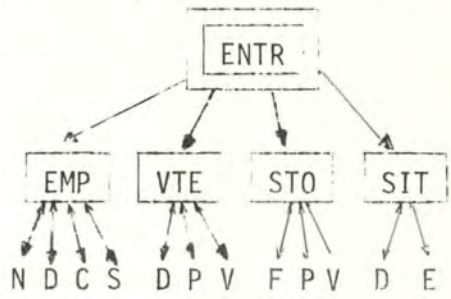
- SEQUEL appartient à cette dernière catégorie de langages mais son originalité est de permettre de sélectionner les données en termes d'ensembles et non de lignes courantes.

La plus simple expression de SEQUEL comporte quatre arguments :
SELECT < plage > FROM < table > WHERE < domaine >< opérateur >< argument > où
< plage > est une liste de noms de colonnes,
< table > est le nom d'une table,
< domaine > est un nom de colonne,
< argument > est une liste d'arguments valorisant le "domaine".

La valeur de cette fonction est donc l'ensemble des valeurs de la liste des colonnes de la table donnée, dont les valeurs des colonnes du domaine, correspondent aux arguments.

Prenons l'exemple d'une base de données décrivant les opérations de vente d'une entreprise.

EMP(NOM, DEPT, CHEF, SAL)
VTE(DEPT, PRO, VOL)
STO(FOURN, PRO, VOL)
SIT(DEPT, ETAGE)



La table EMP donne le nom de chaque employé, son département, son chef et son salaire.

La table VTE fournit le volume annuel des ventes de chaque produit par chaque département.

La table STO délivre le volume annuel des différents produits que livrent chaque fournisseur.

La table SIT indique l'étage auquel est situé chaque département.

Nous donnons aussi le diagramme MSI correspondant.

⊬ Question 1 : trouver les noms des employés du département des jouets .

SEQUEL : SELECT NOM FROM EMP WHERE DEPT = 'JOUET'

ADL : [:EMP(:DEPT='JOUET') [:NOM P]] ou [:NOM(:EMP(:DEPT='JOUET'))]

- La clause WHERE est facultative : son absence revient à restreindre la fonction à une projection de la table entière sur les colonnes choisies.
- Si de plus, la plage et le mot FROM sont omis, la fonction délivre toutes les colonnes de la table choisie.
- Le domaine peut être une expression arithmétique de noms de colonnes, ainsi que la clause SELECT qui peut inclure en outre des fonctions du genre somme, moyenne, maximum, ...
- La clause WHERE a la forme d'une expression booléenne de domaines.

⊬ Question 2 : fournir les noms et les salaires des employés qui travaillent dans le département des jouets et dont le chef est Brioult.

SEQUEL : SELECT NOM, SAL FROM EMP WHERE DEPT = 'JOUET' AND CHEF = 'BRIOULT'

ADL : [:EMP((:DEPT='JOUET') & (:CHEF='BRIOULT'))[:NOM P] ; [:SAL P]]

- Ces fonctions peuvent être composées en appliquant une fonction au résultat d'une autre fonction, qui est donc un critère à remplir par le domaine de la fonction "extérieure".

⊬ Question 3 : trouver les produits vendus par les départements du second étage.

SEQUEL : SELECT PRO FROM VTE WHERE DEPT= SELECT DEPT FROM SIT WHERE ETAGE=2

ADL : [:VTE(:DEPT=DEPT(:1-SIT(:ETAGE=2)))[:PRO P]] ou
[:PRO(:VTE(:DEPT=DEPT(:SIT(:ETAGE=2))) P]

Il faut remarquer que dans ce cas, la fonction extérieure fournit les lignes dont la valeur satisfait le critère portant sur une valeur au moins de la fonction intérieure; SEQUEL permet de lever cette restriction en utilisant le quantificateur ALL dans un critère.

- Les ensembles obtenus par des fonctions peuvent être manipulés par les opérateurs ensemblistes d'union, d'intersection et de différence.

⊬ Question 4 : donner les produits fournis par Fournier et vendus par le département des jouets.

SEQUEL : SELECT PRO FROM STO WHERE FOURN='FOURNIER' ∩ SELECT PRO FROM VTE
WHERE DEPT='JOUET'

ADL : [:PRO((:STO(:FOURN='FOURNIER')) & (:VTE(:DEPT='JOUET')))P]

- L'utilisateur peut aussi construire des n-uples de constantes pour valoriser sa question grâce à la fonction SET.

/ Question 5 : Trouver les départements qui ont parmi leurs employés, Dupont, Dupond et Durant, tous ayant Duval pour chef.

```
SEQUEL : SELECT DEPT FROM EMP WHERE SET(NOM, CHEF)  ⊇
                                {< 'DUPONT', 'DUVAL' >,
                                 < 'DUPOND', 'DUVAL' >,
                                 < 'DURAND', 'DUVAL' >}
ADL    : [ :DEPT((:1-EMP((:NOM='DUPONT') & (:CHEF='DUVAL')))
                 &(:1-EMP((:NOM='DUPOND') & (:CHEF='DUVAL')))
                 &(:1-EMP((:NOM='DURAND') & (:CHEF='DUVAL'))))P ]
```

- SEQUEL permet de qualifier un nom de colonne par un nom de table : ceci se révèle utile lors de la comparaison de lignes de différentes tables portant sur des valeurs de domaines compatibles.

/ Question 6 : donner la liste des lignes ventes et stock qui portent sur le même produit.

```
SEQUEL : SELECT VTE, STO WHERE VTE.PRO=STO.PRO
ADL    : [ :VTE.V1(:PRO=PRO(:1-STO)) P; FOR XØ [ :STO(:PRO=PRO(:V1)) P]]
```

- Enfin, on peut associer une étiquette à une fonction SELECT et s'en servir pour qualifier des colonnes.

/ Question 7 : trouver les noms des employés dont le salaire est plus élevé que celui de leur chef.

```
SEQUEL : B1 : SELECT NOM FROM EMP WHERE SAL  >
                      SELECT SAL FROM EMP WHERE NOM = B1.CHEF
ADL    : [ :NOM(:EMP.B1(:SAL>SAL(:EMP(:NOM=CHEF(:B1))))) P ]
```

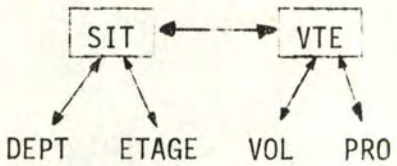
L'étiquette B1 est en fait une variable qui a pour valeur une ligne "employé".

Force est de constater que SEQUEL et ADL ne sont pas entièrement comparables : le premier est un langage conversationnel de recherche dont les questions sont indépendantes alors que le second est plus structuré en vue de l'élaboration d'un programme et qu'il inclut des actions de manipulation. Cela dit, nous restreindrons la comparaison à la désignation.

- Dans beaucoup de questions, les expressions de SEQUEL et d'ADL ne diffèrent que par les mots-clés.
  D'une part, SEQUEL a l'avantage de proposer une forme unique et claire SELECT FROM WHERE; on peut juste lui reprocher sa longueur d'écriture en cas de composition; d'autre part, on peut dire qu'ADL exprime les mots-clés FROM et WHERE sous la même forme d'un critère de relation.
- Dans certaines de ses possibilités, SEQUEL est un langage plus riche puisqu'il permet l'emploi d'opérateurs arithmétiques, statistiques et ensemblistes.
- SEQUEL est aussi plus complet en ce qui concerne l'édition de résultats groupés : il arrive même que, dans le même but, ADL fasse appel à des accès imbriqués là où ce n'est pas intrinsèque au problème (comme en témoigne Q6).
- Dans des questions plus complexes, ADL prend l'avantage sur SEQUEL.

En effet, nous avons vu que le MSI était plus riche que le modèle relationnel; par conséquent, certaines expressions d'ADL seront plus concises que leur équivalent en SEQUEL car les liens privilégiés entre objets, présents dans le MSI et absents dans le modèle relationnel, devront être "construits" en SEQUEL et simplement cités en ADL.

Ainsi, on pourrait définir, entre autres, une relation entre les départements et leurs ventes (c'est-à-dire, les ventes qui ont même département), ce qui donne la partie suivante du diagramme MSI



Reprenons la question Q3 : trouver les produits vendus par les départements du second étage.

SEQUEL la posait ainsi : SELECT PRO FROM VTE WHERE DEPT=SELECT DEPT FROM SIT WHERE ETAGE=2

ADL l'exprimait de la même façon dans l'ancienne structure; dans la nouvelle, ADL permet de la formuler comme suit :

[:PRO(:VTE(:SIT(:ETAGE=2))) P]

4.3. SOCRATE

Les données élémentaires sont appelées caractéristiques dans SOCRATE; un bloc est le regroupement de plusieurs caractéristiques ou même d'autres blocs. Une entité est un bloc qui peut avoir plusieurs réalisations.
Entités et blocs permettent donc de décrire une structure arborescente.
Cependant SOCRATE étend ce modèle hiérarchisé par le concept de caractéristique (d'une entité) "référence" d'une autre entité (peut-être du même type), qui permet de définir une relation entre ces deux types d'entités.

Enfin, on peut accéder à une entité à partir de ses caractéristiques : celle-ci sera dite discriminante lorsque la relation de cette caractéristique à l'entité est une fonction et rapide dans le cas contraire.

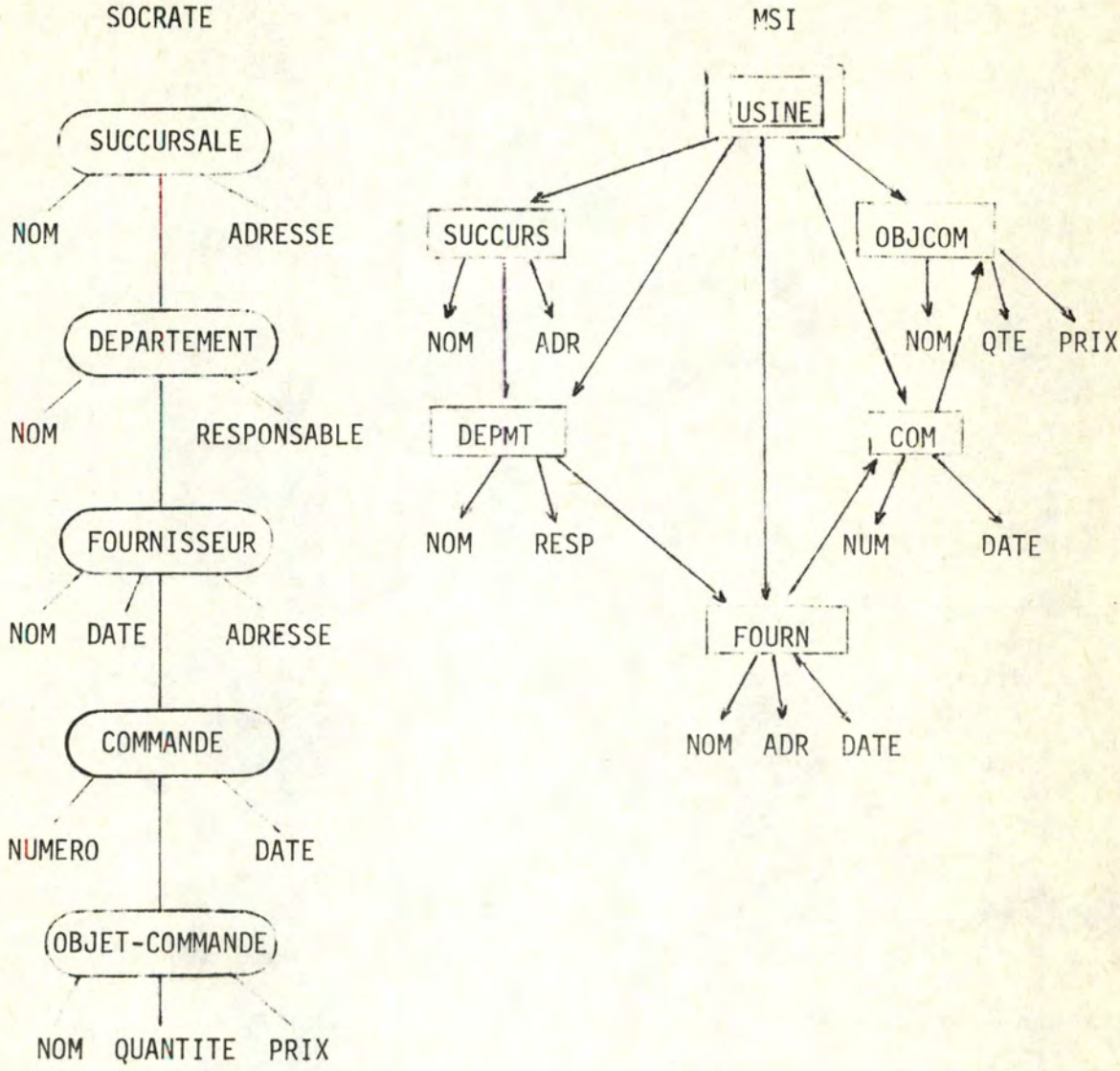
On peut établir la table de correspondance SOCRATE-MSI.

Base de données	- Racine
Entité	- Objet complexe
Bloc	- Objet élémentaire composé
Caractéristique (sauf référence)	- Objet élémentaire
Inclusion de l'entité B dans l'entité A	- Relation (A, B) : Ø-∞, 1-1
Inclusion du bloc B dans l'entité A	- Relation (A, B) : Ø-1, 1-1
Caractéristique référence dans A vers B	- Relation (A, B) : Ø-1, Ø-∞
Caractéristique B de A rapide	- Relation (A, B) : 1-1, Ø-∞
Caractéristique B de A discrimante	- Relation (A, B) : 1-1, Ø-1

SOCRATE propose le graphe de visualisation de la structure d'une BD, suivant .

- Les noeuds de ce graphe sont des caractéristiques : celles-ci seront matérialisées par leur nom, ainsi que les blocs, tandis que les entités seront figurées par un cercle entourant leur nom.
- Les arêtes en trait plein correspondent à des relations hiérarchiques et les arcs orientés de tirets, à des références d'une caractéristique à une entité.

Le premier exemple portera sur une structure purement arborescente.



Le langage de manipulation

Un programme de manipulation est formé d'une suite de requêtes éventuellement groupées (requête POUR) ou conditionnées (requête SI).

Une requête simple est formée d'une commande qui précise l'action demandée, suivie d'une citation qui désigne l'objet sur lequel porte la commande.

Les actions sont du type création, interrogation, suppression, mise à jour.

Exemple : i DATE DE UNE COMMANDE AYANT NUMERO = 25 ; ?

i est la commande d'interrogation.

A. La désignation (ou citation) comporte deux structures fondamentales .

1. Désignation simple

On désigne les entités par le chemin qu'il faut suivre pour y accéder à partir d'un point d'entrée.

1.1. A partir du point d'entrée de la base

Si l'on veut obtenir tous les noms des succursales, on écrira :
i NOM DE TOUTE SUCCURSALE ou POUR TOUTE SUCCURSALE i NOM FIN
Cette seconde forme utilise une requête POUR, qui permet en outre de regrouper des requêtes entre les mots-clés POUR et FIN.

Dans les deux formes, SUCCURSALE est un "qualificateur" de NOM.

On peut aussi ne désigner qu'un élément d'un ensemble; si on veut désigner la première succursale, il faut écrire UNE SUCCURSALE.

1.2. A partir d'une caractéristique

La succursale de nom A peut être désignée par : LA SUCCURS AYANT NOM='A';
Pour ce faire, nous avons utilisé une condition (appelée "filtre").

2. Condition

Le principe consiste à restreindre les collections des éléments par lesquels on passe, dans une désignation, par des expressions booléennes.

2.1. Conditions sur caractéristiques

On désignera toutes les commandes passées avant le 7 avril 77 par :
TOUTE COM AVANT DATE < = '77Ø4Ø7'; DE TOUT FOURN DE TOUT DEPMT DE TOUTE SUCCURS
Les termes AYANT et ";" jouent le rôle de parenthèse ouvrante et fermante.
Il arrive que l'écriture d'un test de caractéristiques se révèle ambiguë de par l'emploi de noms communs; il faut employer des désignateurs assignés à ces entités.

On cherche par exemple toutes les commandes passées à la même date que celle d'enregistrement du fournisseur auquel elles sont adressées et telles que cette date soit postérieure à d :

TOUTE COM X1 AYANT DATE > 'd' ET DATE DE X1 = DATE DE X2; DE TOUT FOURN X2
ADL a repris ce principe et fournit la réponse :
COM.X1((:DATE > 'd') & (:DATE=DATE(:FOURN(:X1))))

2.2. Conditions sur entités

- SOCRATE introduit les quantificateurs existentiels et universel qui sont EXISTE (ou son contraire PAS) et TOUT.
  Pour obtenir les fournisseurs ayant au moins une commande, on écrira :

  TOUT FOURN AYANT EXISTE COM; DE TOUT DEPMT DE TOUTE SUCCURS

  On recherche encore tous les fournisseurs dont toutes les commandes ont été passées à une certaine date d.

  TOUT FOURN AYANT TOUTE COM AYANT DATE='d';; DE TOUT DEPMT DE TOUTE SUCCURS

  De manière analogue, ADL traduira cette condition par un critère de relation de quantificateur ALL : [:FOURN(:ALL COM(:DATE='d'))P]

- Une condition générale aura la forme d'une expression booléenne de conditions sur des caractéristiques et de conditions sur des entités, à quelque niveau que ce soit.

  Voici un exemple reprenant les formes vues jusqu'à présent : on veut obtenir tous les fournisseurs ayant passé une commande au moins sur des pommes, avant le 7 avril, et dont l'adresse est A :

```
TOUT FOURN AYANT EXISTE UNE COM AYANT DATE <='770407'
                                    ET NOM DE OBJCOM='POMME';
              ET ADRESSE='A'; DE TOUT DEPMT DE TOUTE SUCCURS
[ :FOURN((:1- COM((:DATE ¬1> '770407')
                  &(:OBJCOM(: NOM='POMME')))
        &(: ADR='A'))                              P]
```

B. Voyons les principales autres requêtes.

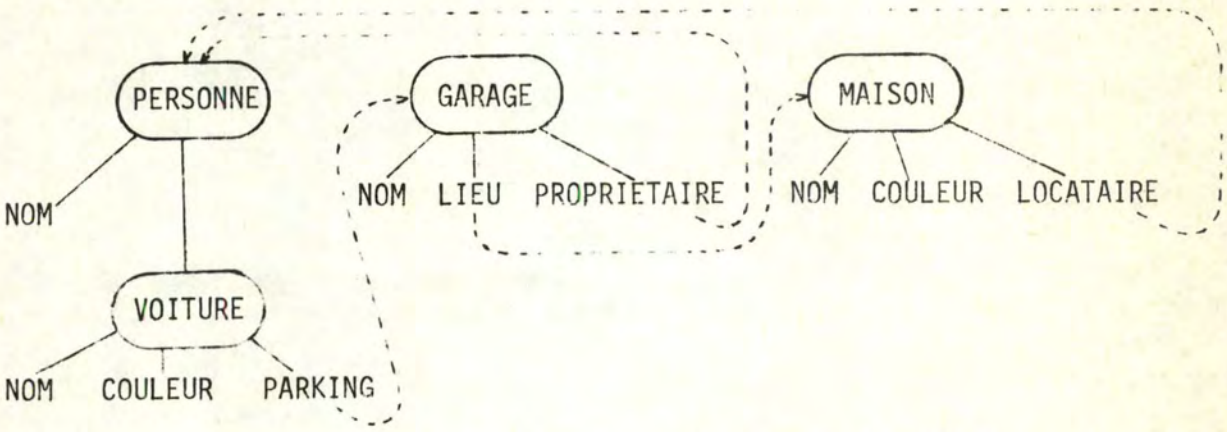
- La requête SI ... ALORS ... SINON ... FIN est semblable à l'instruction conditionnelle d'ADL.
- La requête POUR ... FIN correspond à une mise en évidence de qualificateurs.
  Par exemple, la requête POUR TOUTE SUCCURS

```
                    i NOM
                    i ADR
               FIN
```

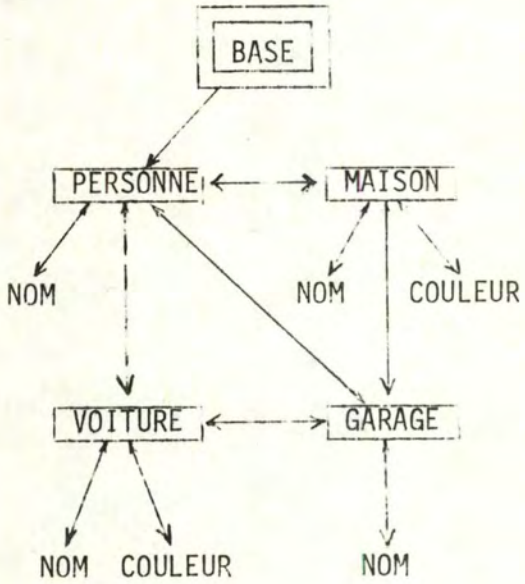
  met le quantificateur SUCCURS en évidence .
- La requête DEPUIS permet de commencer une boucle portant sur des entités à partir d'un numéro de réalisation explicité ou contenu dans une variable entière.
  On imprime ainsi le nom de toutes les succursales à partir de la 5°, par :
  DEPUIS 5 i NOM DE TOUTE SUCCURS ?

Considérons un autre système réel, donnant lieu à une structure en réseau : des personnes possèdent des voitures qu'ils rangent dans des garages dont ils peuvent être propriétaires; ces garages font partie de maisons louées par des personnes.

Le modèle SOCRATE correspondant sera :



En tenant compte de la table de correspondance, voici le MSI équivalent :



Question 1 : On considère les personnes dont les voitures ont la couleur des maisons dont elles sont locataires et on veut désigner le nom de ces voitures.

SOCRATE : NOM DE VOITURE AYANT COULEUR = COULEUR DE MAISON AYANT LOCATAIRE = PERSONNE EN COURS;; DE PERSONNE

Dans cet exemple, la seconde condition est un critère d'égalité portant sur un nom de caractéristique référence et sur le nom de l'entité référencée : on veut ainsi exprimer que la valeur de cette caractéristique doit donner l'accès à une réalisation de l'entité visée.

Voici la traduction en ADL :

[:PERSONNE.P1[:VOITURE(:COULEUR=COULEUR(:MAISON(:P1)))[:NOM P]]]

ou VOITURE.V1(:COULEUR=COULEUR(:MAISON(:PERSONNE(:1-V1))))

Question 2 : Désigner toute personne locataire du lieu de parking d'une de ses voitures et propriétaire de ce parking à condition que la couleur de ce garage soit celle de sa voiture.

SOCRATE : NOM DE PERSONNE AYANT EXISTE VOITURE AYANT LOCATAIRE DE LIEU DE PARKING = PERSONNE EN COURS ET PROPRIETAIRE DE PARKING = PERSONNE EN COURS ET COULEUR DE LIEU DE PARKING = COULEUR DE VOITURE EN COURS

On emploie ici des qualificateurs qui sont des références : de façon fort intuitive, on substitue un nom d'entité par un nom de caractéristique qui la référence.

Ainsi LOCATAIRE DE LIEU DE PARKING signifie LOCATAIRE DE MAISON ≡ LIEU DE GARAGE ≡ PARKING

et comme LOCATAIRE réfère PERSONNE, on compare la valeur de cette expression à une réalisation de PERSONNE.

L'utilisation de désignateurs permet d'alléger cette formulation :

NOM DE PERSONNE X1 AYANT UNE VOITURE X2 AYANT LOCATAIRE DE LIEU X3 DE PARKING X4=X1 ET PROPRIETAIRE DE X4=X1 ET COULEUR=COULEUR DE X3;;

En ADL,[:PERSONNE.P1(:VOITURE.V1((:GARAGE((:P1)&(:MAISON(:P1)))
&(:COULEUR=COULEUR(:GARAGE(:V1))))[:NOM P]]

Question 3 : Trouver les personnes dont la couleur d'une voiture est unique : aucune autre personne ne l'a adoptée pour une de ses voitures.

SOCRATE : NOM DE PERSONNE X1 AYANT UNE VOITURE AYANT COULEUR¬=COULEUR DE TOUTE VOITURE DE TOUTE PERSONNE ¬=X1;;

ADL:[:PERSONNE.P1(:1-VOITURE(:COULEUR¬=COULEUR(:ALL VOITURE ¬(:P1))))
[:NOM P]]

Remarques

- Un qualificateur peut être hiérarchiquement inférieur à l'identificateur qualifié : dans ce cas, on emploiera DONT et non DE.
  Ainsi, si X1 désigne une réalisation de VOITURE, on peut écrire NOM DE UNE PERSONNE DONT X1.
- Dans certains cas que nous ne préciserons pas, un filtre doit commencer par TELQUE et non par AYANT.

En conclusion, SOCRATE et ADL sont très proches en ce qui concerne la recherche de données : ces langages reposent sur la notion de navigation dans un réseau de relations.

- Sur le plan du modèle de structure de l'information, SOCRATE se particularise par une double démarche qui consiste à percevoir des données hiérarchisées dans un monde réel puis à établir des liens entre ces hiérarchies par l'intermédiaire de données communes. Les différents arbres sont alors mêlés en un réseau.
- Cette différence de conception entre les relations hiérarchiques et les relations par référence se traduit malheureusement par des utilisations différentes dans des désignations.
- De plus, SOCRATE oppose au symbolisme graphique d'ADL, une forme plus "naturelle", dans les conditions notamment, ce qui ne va pas sans une prolixité certaine, voire une certaine complexité (nous en voulons pour preuve la question Q2).
- Toujours à propos de l'emprunt de relations de passage, on dispose dans SOCRATE de trop de mécanismes pour exprimer la même chose : (POUR, DE, DONT, AYANT, AYANT EXISTE, TELQUE); ADL se contente de la boucle d'accès (qui correspond à POUR) et du critère de relation (qui reprend toutes les autres formes).
- L'emploi de quantificateurs est plus limité chez SOCRATE que dans ADL. C'est si vrai que le quantificateur UN, du premier langage, possède deux significations : devant un qualificateur, il désigne la première réalisation de cette entité et dans un filtre, il impose au filtre d'agir sur une réalisation au moins.

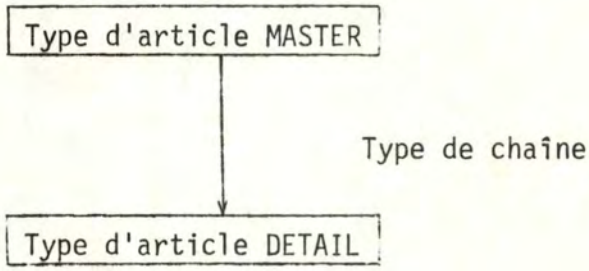
4.4. IDS1.

4.4.1. Principes fondamentaux

L'unité de données en IDS est le "record" (que nous appellerons aussi article). Les records sont chaînés entre eux : une chaîne possède un seul record MASTER qu'elle relie à plusieurs records DETAIL.

Tout schéma de fichier IDS est représentable sous la forme d'un graphe dont :

- les sommets sont des types de records;
- les arcs sont des types de chaînes.



La table de correspondance MSI-modèle IDS serait trop longue à détailler; elle a été vue en détail dans le mémoire de F. BASTIN [10] .

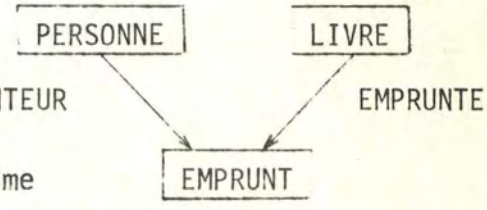
Il nous suffit de savoir que :

- la chaîne IDS est une relation one to many "forte" (Ø-∞, 1-1),*
- le record IDS correspond à une réalisation d'objet complexe,
- une relation one to one peut être représentée par un type de chaîne à un seul article détail,
- une relation many to many faible (Ø-∞, Ø-∞) sera figurée par 2 chaînes qui se recoupent en un article d'intersection.

  Tel serait le cas d'une relation (PERSONNE, LIVRE) dans l'exemple de la bibliothèque, cité lors de l'étude du DML de l'Institut.

  Une personne peut emprunter plusieurs livres et un livre peut être emprunté par plusieurs personnes.

  C'est pourquoi, nous avons créé le type d'article EMPRUNT, dont un article exprime l'association entre un article PERSONNE et l'article LIVRE emprunté.



---

* Une relation de caractéristiques (I-J, K-L) est dite forte ssi K > Ø.
Si K est nul, cette relation sera dite faible.

4.4.2. Langage de manipulation

Le principe d'un tel langage est de cheminer dans l'ensemble des articles liés par les relations formées par les chaînes.
A chaque type d'article, est associée une zone standard, décrite dans une DATA DIVISION, pouvant contenir un article de ce type : chaque fois qu'un article est désigné, il peut être placé dans cette zone, qu'on peut donc considérer comme l'article "courant" d'un tel type.

Voyons comment un article peut être désigné :

A. Indépendamment des opérations précédentes

La primitive réalisant cette fonction est RETRIEVE < type d'article > RECORD.
Le programmeur devra, préalablement à cet appel IDS, garnir toutes les zones identifiantes (MATCH-KEY) des types d'article jalonnant le chemin de recherche et ce, à l'aide d'instructions du langage hôte COBOL.

B. En fonction des opérations précédentes

Ce genre de désignation est basé sur un parcours "pas à pas" d'une chaîne.
IDS offre plusieurs possibilités dont voici les principales :

B.1. Obtenir l'article qui suit l'article courant d'une chaîne

RETRIEVE NEXT RECORD OF < type de chaîne > CHAIN.
L'intérêt de cette primitive est d'accéder à une série d'articles détails reliés à un article maître, obtenu par un positionnement préalable établi par une primitive RETRIEVE ... RECORD.
C'est donc au programmeur qu'il échoit d'écrire une telle boucle de traitement, en incluant les branchements nécessaires en cas d'échec ou de repositionnement, en fin de chaîne, à l'article maître original.

B.2. Obtenir l'article maître de l'article courant d'une chaîne

RETRIEVE MASTER RECORD OF < type de chaîne > CHAIN.
RETRIEVE HEAD RECORD OF < type de chaîne > CHAIN.
La première de ces primitives rend courant l'article maître alors que la seconde fournit simplement l'article maître tout en conservant l'article courant.

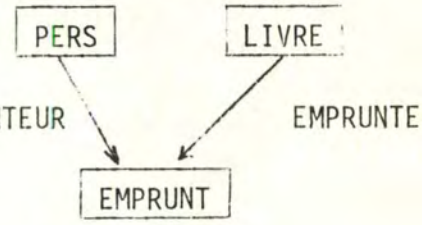
B.3. Obtenir le premier article détail d'une chaîne

Il suffit de combiner 2 des opérations vues, RETRIEVE MASTER RECORD OF ..CHAIN
RETRIEVE NEXT RECORD OF ...CHAIN.

Les opérations de manipulation sont la suppression et la modification, qui touchent le courant d'un type d'article tandis que l'insertion d'un article impose de garnir préalablement toutes les MATCH-KEY du chemin d'insertion ou encore de se positionner sur les articles maitres correspondants, par une des actions RETRIEVE.
La primitive STORE < nom d'article > RECORD assure l'insertion d'un article.

4.4.3. Exemples

Reprenons l'exemple de la bibliothèque, dont voici le schéma IDS :



Il n'est pas possible de définir une relation faible d'accès aux livres disponibles : nous supposerons qu'il existe une caractéristique DISP de LIVRE, qui est un indicateur booléen.

Programmons l'applications vue : l'emprunt d'un livre.

```
ENTER IDS.
        OPEN.
ENTER COBOL.
MOVE N TO NUM OF LIVRE.
ENTER IDS.
        RETRIEVE LIVRE RECORD IF ERROR GO TO FIN
                                    ELSE MOVE TO WORKING-STORAGE .
ENTER COBOL.
IF DISP OF LIVRE=Ø THEN DISPLAY "LIVRE INDISPONIBLE" UPON TERMINAL
                    GO TO FIN.
MOVE X TO NUM OF PERS.
ENTER IDS.
        RETRIEVE PERS RECORD IF ERROR GO TO CREAT-PERS
                                   ELSE GO TO PERS-EXIST.
ENTER COBOL.
CREAT-PERS.
          MOVE X TO NUM OF PERS,
               Y TO NUM OF PERS,
               Z TO ADR OF PERS.
          ENTER IDS.
                    STORE PERS RECORD.
          ENTER COBOL.
          GO TO CREAT-EMPR.
PERS-EXIST.
          ENTER IDS.
                    RETRIEVE NEXT RECORD OF EMPRUNTEUR CHAIN
                    MOVE TO WORKING STORAGE.
          ENTER COBOL.
          IF DATE-RET OF EMPRUNT=Ø THEN DISPLAY "EMPRUNT REFUSE" UPON
                                    TERMINAL GO TO FIN.
CREAT-EMPR.
          MOVE Ø TO DATE-RET OF EMPRUNT.
          MOVE TODAYS-DATE TO DATE-EMP OF EMPRUNT.
          ENTER IDS.
                    STORE EMPRUNT RECORD.
          ENTER COBOL.
FIN.
```

Si l'on compare cet exemple à son équivalent en ADL, vu dans le chapitre 3, on perçoit combien l'approche d'ADL est différente de par :

- son instruction de boucle d'accès,
- son instruction conditionnelle qui comporte des accès implicites,
- son instruction de création, qui permet de spécifier les caractéristiques de la nouvelle réalisation ainsi que ses liens avec d'autres objets complexes.

Il faut reconnaître que cette application convient bien aux langages procéduraux comme IDS, et non aux langages de boucles, comme ADL.

Ainsi, si l'on veut traiter tous les emprunts faits jusqu'à une date donnée par une personne donnée, les différences entre séquences IDS et ADL seront plus marquées.

```
IDS. ENTER IDS.
               OPEN.
     ENTER COBOL.
     MOVE X TO NUM OF PERS.
     ENTER IDS.
               RETRIEVE PERS RECORD IF ERROR GO TO ERR-PERS.
     ENTER COBOL.
     LOOP.
          ENTER IDS.
                    RETRIEVE NEXT RECORD OF EMPRUNTEUR CHAIN
                    MOVE TO WORKING-STORAGE
                    IF PERSONNE GO TO EXIT. *
          ENTER COBOL.
          IF DATE-EMP OF EMPRUNT > Y GO TO LOOP.
          < traitement des emprunts >
          GO TO LOOP.
     EXIT.
        ....
     ERR-PERS.
               DISPLAY "PERSONNE INEXISTANTE" UPON TERMINAL.
     ENTER IDS.
               CLOSE.
     ENTER COBOL. ...

ADL. ENTER BIBLIO.B1
     IF(B1(:PERS(:NUM=X))) THEN
       REACH:ALL EMPRUNT((*:PERS(:NUM=X))&(:D-E < Y))
                         < traitement >
       END
                           ELSE < écrire "personne inexistante" > END
     EXIT
```

* Ce test prévoit le cas où tous les emprunts ont été parcourus.

4.5. IMS

4.5.1. Principe de base

Le groupe élémentaire d'informations est appelé segment. Les divers segments d'une structure IMS sont répertoriés par types et référencés par leur nom. Les segments sont reliés entre eux pour éviter la redondance des informations.

Une structure IMS repose sur la définition de plusieurs data bases (fichiers physiques) interconnectés.

4.5.2. Structure logique

On peut distinguer trois phases successives dans la conception d'une structure IMS.

1. Structure hiérarchique d'une data base

Les segments d'une DB physique forment une structure arborescente :
- à chaque noeud correspond un type de segment (et donc des données),
- chaque arc représente une relation hiérarchique entre deux types de segments.

Nous parlerons de relation physique entre un segment parent physique et un segment enfant physique, par analogie à la structure hiérarchique définie.

2. Relations logiques

On peut établir d'autres relations hiérarchiques entre deux types de segment appartenant à la même BD ou à deux BD différentes; IMS parle alors de relation logique entre segment enfant logique et segment parent logique.
Tout comme avec SOCRATE, nous sommes partis d'une structure hiérarchique des données, que l'on complète en un réseau au moyen de relations logiques (références).

3. Vue logique d'un programme sur une structure (Program Communication Block)

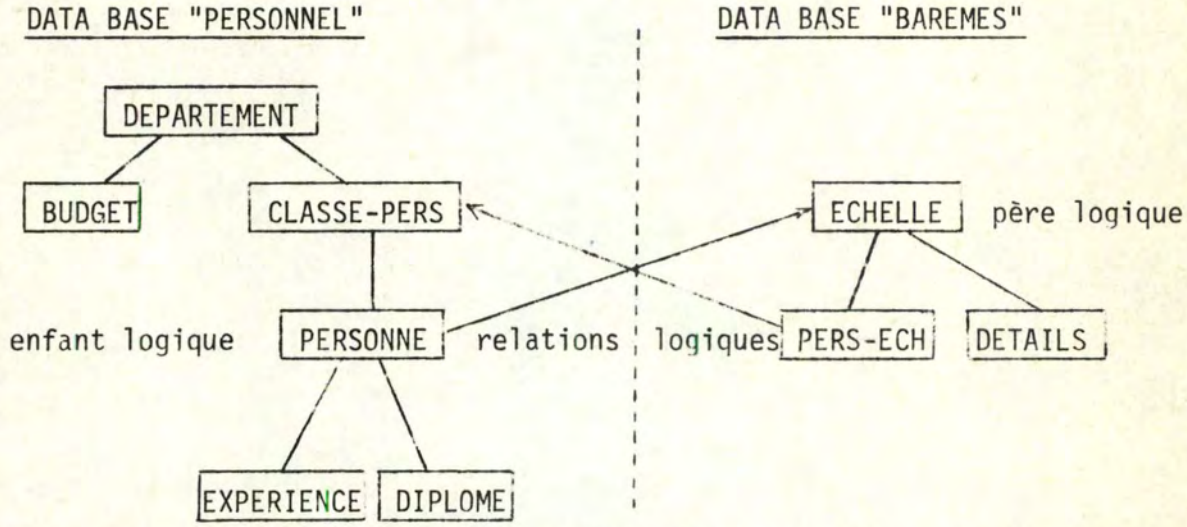
IMS présente un ensemble de data bases à un programme, en remodelant cette structure sous une forme arborescente. Cette troisième étape peut paraître nous ramener au point de départ; il n'en est rien car c'est en fonction des besoins de l'utilisateur que cette restructuration est entreprise.

A. Le programmeur peut n'avoir besoin que d'une vue partielle de l'ensemble des BD, adaptée à un programme de traitement spécifique.
B. Au contraire, le programmeur peut avoir une vision globale des DB en une seule arborescence.

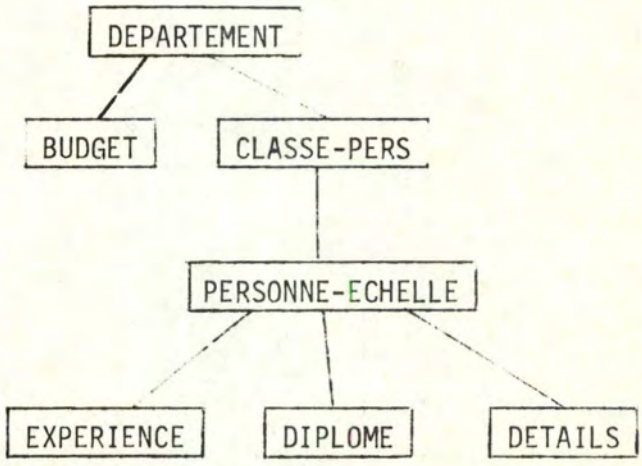
Pour ce faire, IMS use des règles de déduction suivantes :

- le type de segment racine de la nouvelle structure doit être le type de segment racine d'une des DB physiques,
- à tout segment enfant logique de la DB choisie comme racine, peut être juxtaposé son parent logique,
- les enfants physiques (ou même les parents physiques) de ce dernier segment deviennent les enfants physiques de l'entité segment enfant logique - parent logique ainsi formée.

Voici un exemple de structure IMS.



Si le programmeur prend pour racine la DB Personnel, IMS lui présentera le schéma :



N.B. : Le type de segment PERS-ECH a disparu car c'était un segment fictif qu'il avait fallu rajouter dans la BD Barêmes pour respecter la règle suivante d'IMS : un segment ne peut être à la fois enfant logique et parent logique.

Il importe de remarquer qu'un tel schéma n'est qu'une vision de la réalité; autrement dit, il est assimilable au schéma externe préconisé par le rapport ANSI-SPARC.
Il reste à voir comment on peut déclarer une telle structure logique.
A chaque programme est associé un PROGRAM SPECIFICATION BLOCK(PSB) qui regroupe un ensemble de blocs appelés PROGRAM COMMUNICATION BLOCK(PCB) : chacun de ceux-ci décrit une structure logique d'arbre que le programme va traiter.

Un PCB repose sur une DATA BASE DESCRIPTION (DBD) qui peut être physique ou logique; une DBD physique est la formalisation d'une structure hiérarchique alors qu'une DBD logique est l'interrelation d'une ou plusieurs DBD physiques.

4.5.3. Langage de manipulation (DL/1)

Un programme d'utilisateur travaille sur une structure particulière d'arbre, spécifiée par un nom de PCB. L'utilisateur parcourt dans cette structure un chemin correspondant à une branche de cet arbre.

La structure d'un programme est identique à celle d'un programme IDS : un langage hôte accueille les manipulations élémentaires, qui sont des appels à un sous-programme dont les arguments sont :
- le code fonction,
- le nom du PCB correspondant à la vue désirée de l'ensemble des DB,
- parfois, une ou plusieurs conditions portant sur des segments, appelées "segment search arguments" (SSA).

Voici un résumé des opérations du DL/1, avec leur code.

GET UNIQUE(GU)	recherche directe
GET NEXT(GN)	recherche séquentielle (après un GU)
GET NEXT WITHIN PARENT(GNP)	recherche séquentielle des fils d'un parent
GET HOLD(GHU, GHN, GHNP)	idem mais à employer avant DLET ou REPL
INSERT(ISRT)	insertion d'un nouveau segment
DELETE(DLET)	suppression d'un segment existant
REPLACE(REPL)	modification d'un segment existant

Un SSA consiste en un nom de segment, suivi éventuellement d'une condition. Si la condition est absente, toute occurrence du segment cité satisfera ce SSA (pour autant qu'elle fasse partie du chemin hiérarchique défini par les autres SSA).

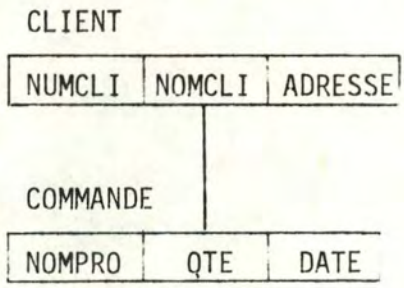
S'il y a une condition, elle est formée d'expressions reliées par les opérateurs booléens "and" et "or"; chaque expression compare un champ (caractéristique) du segment à une valeur alphanumérique.

Le principe d'emploi des SSA est le suivant :

- les opérations GU et ISRT requièrent des SSA spécifiant le chemin hiérarchique tout entier, depuis la racine;
- les opérations GN et GNP peuvent ne pas comporter de SSA mais si elles le font, ces SSA doivent décrire un chemin hiérarchique (qui ne doit pas nécessairement partir de la racine),
- les opérations DLET et REPL ne sont pas accompagnées de SSA.

Reprenons l'exemple présenté p.47, des clients d'une entreprise et de leurs commandes.

Supposons qu'on nous présente une telle BD logique.



Posons-nous deux questions .

Question 1 : trouver les noms des produits commandés par le client 274.

```
     GU CLIENT(NUMCLI='274')
NEXT.GNP COMMANDE
     trouvée ? si non, aller à EXIT
     PRINT NOMPRO
     aller à NEXT
EXIT. ...
```

Question 2 : trouver les numéros des clients qui ont commandé le produit 472.

```
     GU CLIENT
NEXT.GN CLIENT
     trouvé ? si non, sortir
     GNP COMMANDE(NOMPRO='472')
     trouvée ? si non, aller à NEXT
     PRINT NUMCLI
     aller à NEXT
EXIT. ...
```

Bien que ces deux questions soient symétriques, en ce sens que l'une décrit un chemin de recherche inverse de celui de l'autre, leurs réponses ne le sont pas. La raison en est que le DL/1 privilégie les recherches dans le sens parent-enfant : il ne possède pas de primitive de recherche du parent d'un enfant.

4.5.4. Conclusions

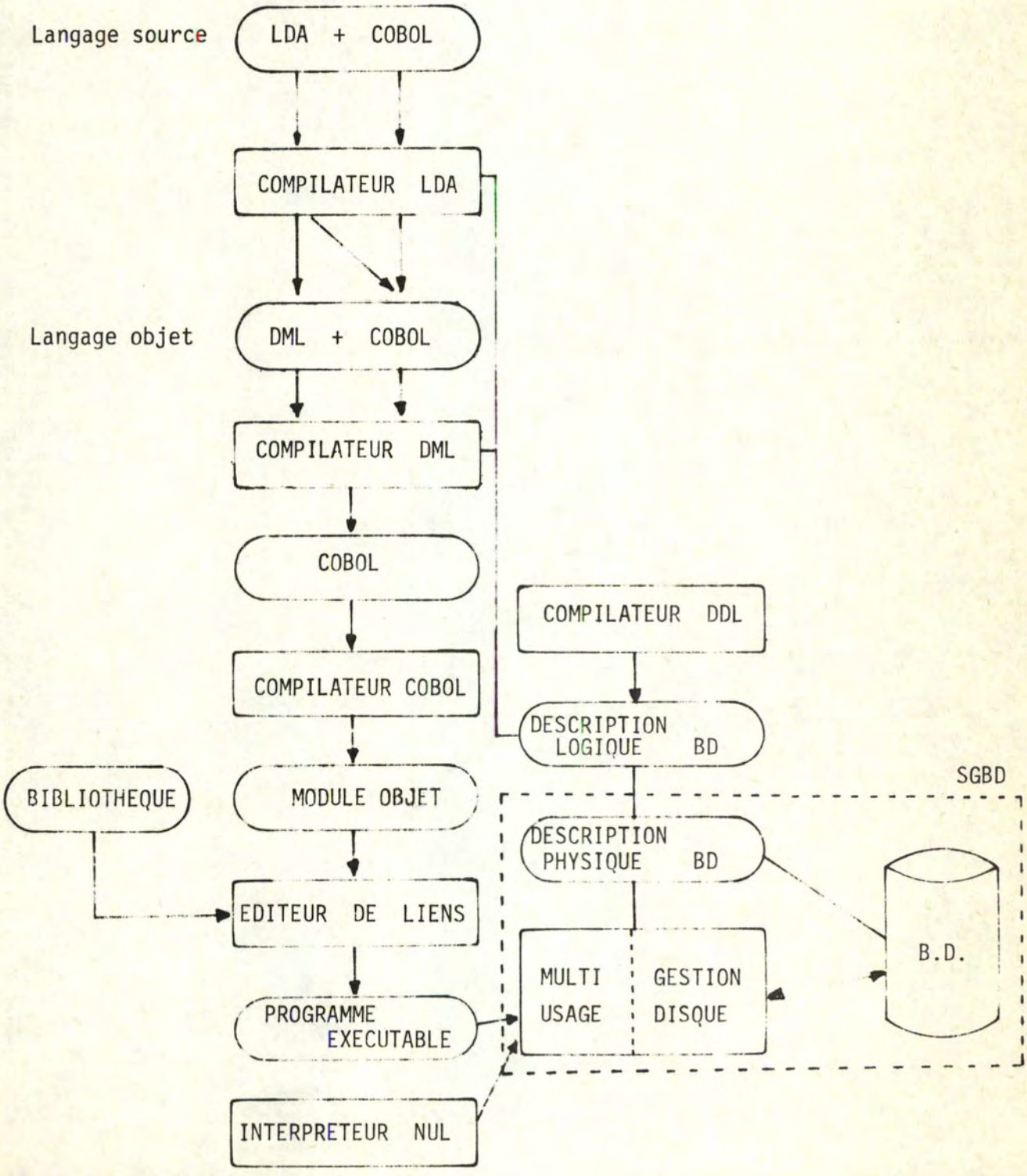
- Le modèle de structure de l'information d'IMS est riche puisqu'il donne la possibilité à l'utilisateur de percevoir plusieurs aspects de la réalité en fonction de son problème.
- Les possibilités de construire un réseau de relations dans ce modèle sont moins grandes que dans le MSI, le modèle d'IDS et même dans le modèle de SOCRATE , qui est basé sur le même principe . Une des règles de construction d'une BD logique est en effet, qu'un segment enfant logique doit avoir un seul parent physique et un seul parent logique; autrement dit, un type de segment ne peut être "origine" de plus d'une relation logique.
- Le langage de manipulation est pauvre : il est réservé à un programmeur averti quant à sa forme et il ne permet pas de parcours de graphe aussi complexes que ce qui est possible en IDS, par exemple.

CHAPITRE 5 : Le compilateur du LDA

5.1. Le système d'exploitation de bases de données

Ce système présenté par l'équipe "Grands Fichiers" de l'Institut d'Informatique, comprend :

- un sous-système de gestion de BD physiques;
- une fonction de description logique de données, assurée par le DDL;
- une fonction de manipulation de ces données, par l'intermédiaire du DML et du LDA; il existe en outre un interpréteur NUL.



A ce stade de son évolution, le LDA n'est pas encore un langage indépendant : ses commandes sont insérées dans des instructions COBOL afin de disposer des opérateurs arithmétiques nécessaires aux applications de gestion.

Les instructions du LDA seront répérées par un caractère spécial préfixé pour les distinguer des instructions COBOL, qui seront simplement copiées sur un fichier temporaire.

Le compilateur du LDA ne traitera que les instructions préfixées et les transformera en séquences formées de macros du DML et d'instructions du COBOL.

5.2. Le compilateur ADL

L'analyseur syntaxique fait appel à l'analyseur lexicographique, qui lui communique des symboles c'est-à-dire des ensembles signifiants de caractères.

L'analyseur syntaxique se compose de différents modules : l'un d'entre eux, le module directeur, fait appel au module déterminé par l'analyse du symbole fourni. Chacun de ces modules vérifie la syntaxe du flux de symboles en fonction du schéma externe (*) de chacune des BD manipulées.

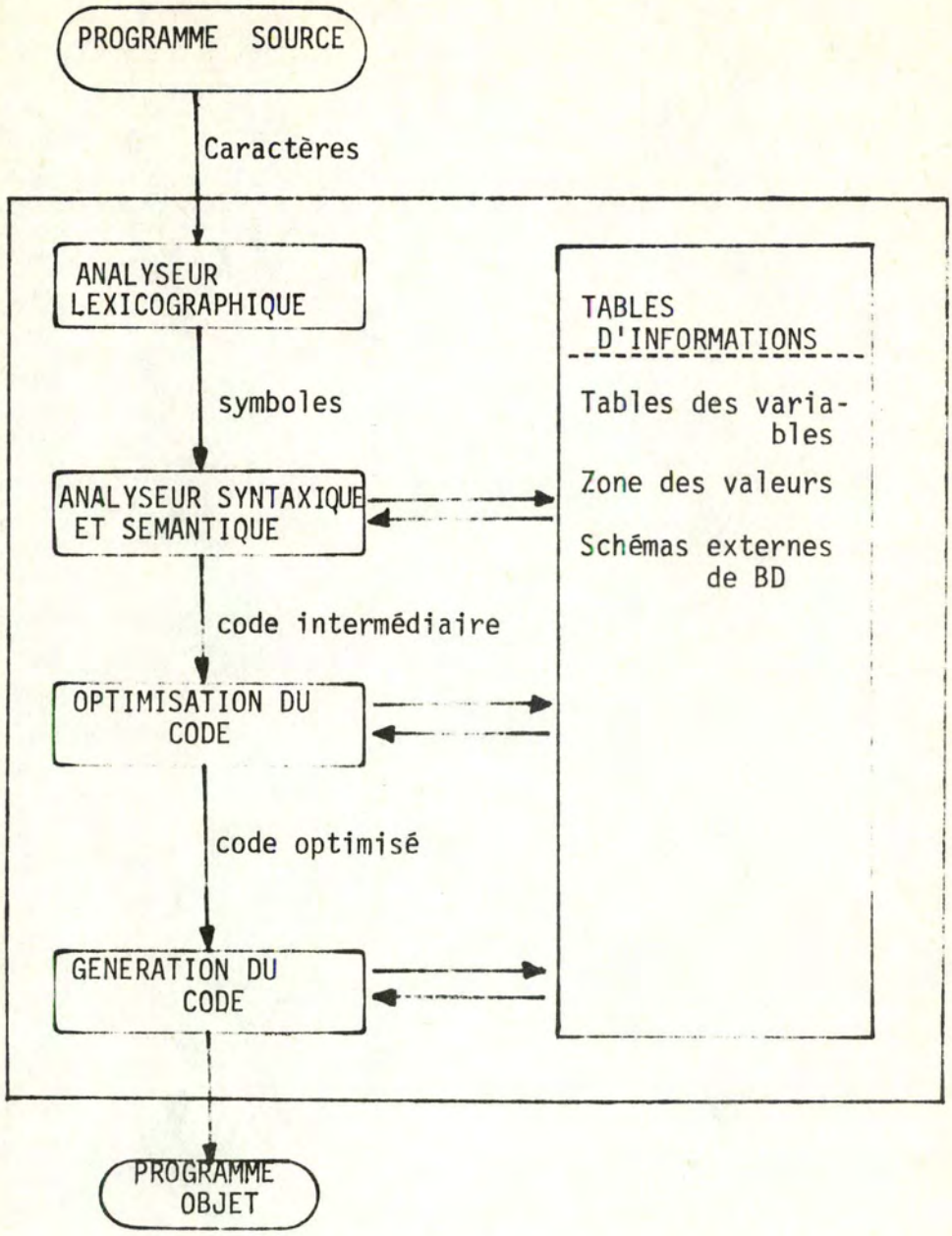
Ces modules garnissent aussi les tables des variables de désignation et de travail, ainsi que la zone des valeurs.

Enfin, ils activent des routines sémantiques, qui visent à créer le code intermédiaire (CI) correspondant au flux de symboles analysés.

Lorsque le CI a été construit dans son entièreté, c'est-à-dire au moment où l'analyseur lexicographique n'a plus de caractère à traiter, le module directeur appelle la routine de génération du code objet, qui travaille sur le CI et les tables d'informations.

La figure suivante résume ces différentes fonctions.

(*) Le schéma externe est la description logique d'une BD, fournie par le DDL.



5.3. Génération du code intermédiaire

5.3.1. Pourquoi employer un code intermédiaire ?

Souvenons-nous que le modèle MSI permet d'intégrer en un formalisme, la description de relations sémantiques entre données et la description de relations d'accès entre ces données.

De même, un programme ADL peut être interprété au niveau sémantique et au niveau des accès; par conséquent, plusieurs désignations peuvent avoir la même interprétation sémantique tout en impliquant des accès différents.

Ainsi, un utilisateur voulant obtenir la liste des noms de tous les professeurs de la Faculté de Lettres, peut formuler sa question de 3 façons différentes :

1. Il peut commander d'accéder à tous les employés des FNDP et ne retenir parmi ces personnes, que celles qui professent dans la faculté de nom LETTRES :
   [: PERSONNE(professeur : FACULTE(:NOM='LETTRES'))[:NOM P]]
2. Il peut vouloir accéder à toutes les facultés; dès qu'il obtient la faculté des Lettres, il accède à tous ses professeurs :
   [: FACULTE(:NOM='LETTRES')[professeur:PERSONNE[:NOM P]]]
3. Il peut enfin n'accéder qu'à la faculté de Lettres, puis à ses professeurs :
   NOM='LETTRES'[: FACULTE [professeur : PERSONNE[:NOM P]]]

Supposons qu'il y ait 6 facultés et en moyenne, 100 personnes dont 10 professeurs, par faculté.

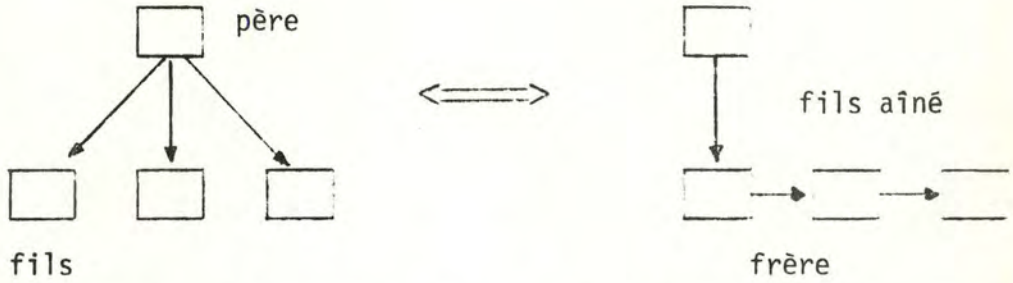
La 1° désignation commandera d'effectuer 6 * 100/2 accès en moyenne, c'est-à-dire 300; la 2° en comptera 6/2 + 10, c'est-à-dire 13; la 3° se réduira à 1 + 10, c'est-à-dire 11 accès.

Cette estimation est très grossière : pour être réaliste, il faudrait tenir compte de la répartition des enregistrements sur les pages de fichiers, de la concurrence des programmes... . Ce nonobstant, on voit l'influence que peut avoir une simple permutation - "passer par personne" puis "par faculté" et inversément - des termes du langage source et donc du CI.

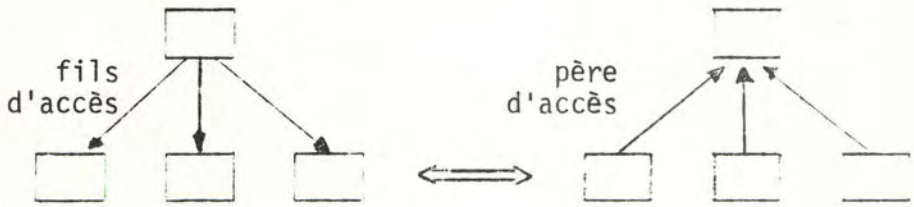
La raison d'existence de ce code est ainsi de pouvoir remodeler certaines questions formulées inefficacement du point de vue des accès sous-jacents; c'est à cette tâche qu'est dévolu un module d'optimisation, sur base d'estimations statistiques, dont l'élaboration dépasse le cadre de ce mémoire.

Le CI doit donc être assez souple pour être manipulé et il doit traduire une structure de séquences d'instructions; c'est pourquoi nous avons adopté une structure arborescente comme forme interne d'un programme ADL.

- D'une façon générale, un noeud de l'arbre de programme correspond à un contexte, c'est-à-dire une collection de réalisations tandis que ses descendants représentent la séquence d'actions à effectuer sur cette collection. Nous avons choisi de construire des arbres binaires afin d'équilibrer les longueurs des représentations des différents types de noeuds et afin d'éviter de remonter au père lorsqu'on parcourt ses fils.



- Rappelons que le CI doit aussi traduire une arborescence d'accès; on pourrait imaginer l'emploi d'un ou de plusieurs pointeurs "fils d'accès" à cette fin mais le soin de ne pas allonger certains noeuds, nous commande de prendre la convention inverse : nous n'userons que d'un pointeur "père d'accès" (noté FA).

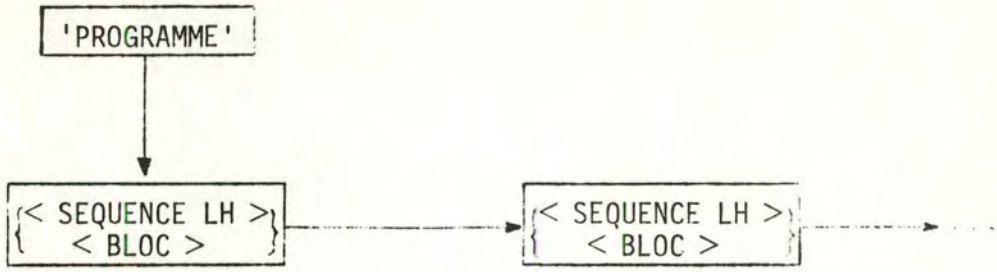


- Un noeud particulier permet de référer une entrée de la table des VD : nous l'appellerons "noeud GENERATE" par analogie à l'instruction du même nom. Dans certaines formes de désignation (comme les actions d'accès, le critère de relation, ...), l'utilisateur peut assigner explicitement une réalisation atteinte à une VD ou il peut ne pas le faire; dans tous les cas, nous avons choisi de généraliser la construction d'un noeud GENERATE : l'entrée visée de la table des VD contient soit le nom d'une VD citée explicitement, soit le nom d'une VD générée par le compilateur, dans le cas contraire.

Comme la génération du code objet de cette pseudo-instruction GENERATE sera différente de celle de l'instruction GENERATE proprement dite, nous aurons deux types de noeuds GENERATE.

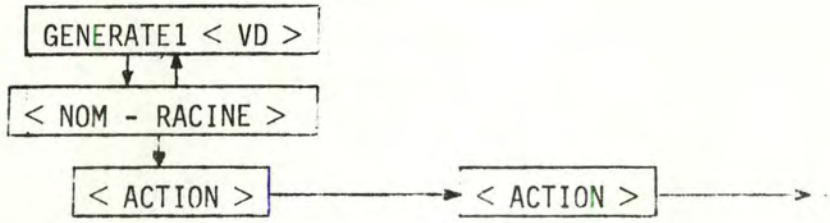
5.3.2. Code intermédiaire associé aux différentes productions

5.3.2.1. Le programme



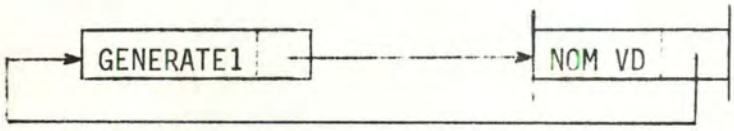
Le compilateur ADL ignore les séquences du langage hôte : il les recopie l'une après l'autre sur un fichier, sans opérer de vérifications; c'est pourquoi le noeud < SEQUENCE LH > contiendra une plage de numéros de lignes, qui renvoient aux enregistrements correspondants du fichier.

5.3.2.2. Le bloc



Nous considérons que cette action est une forme particulière d'accès : c'est pourquoi nous associons un noeud GENERATE1 au nom de la racine.

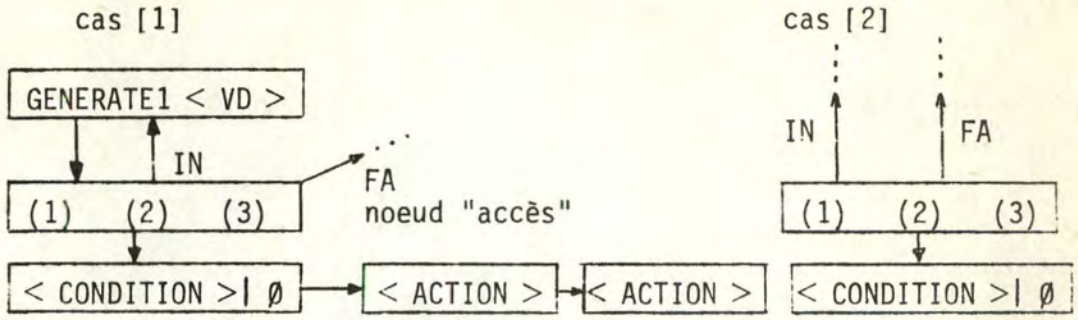
- Le noeud GENERATE1 (correspondant à une pseudo-instruction GENERATE) contient un pointeur vers la table des VD.



Une entrée de la table des VD fournit le nom de la variable et entre autres, un pointeur vers le noeud GENERATE associé.

- Le noeud < nom racine > contient :
  . un pointeur vers le noeud GENERATE associé (on verra que ce n'est pas toujours le père);
  . un pointeur vers l'enregistrement du schéma externe correspondant au nom de la racine.
- Le noeud < action > figure un ensemble de noeuds représentant une action parmi celles décrites aux pages suivantes; rappelons que le bloc et la séquence LH sont aussi des actions.

5.3.2.3. Les actions d'accès et d'accès fictif



- Le noeud "accès" contient les informations suivantes :
  - (1) : un pointeur vers le nom de la relation d'accès empruntée, dans le schéma externe (ce pointeur est nul dans le cas d'une relation à partir d'une racine, et il est absent pour un accès fictif)
  - (2) : un ensemble d'informations qui définissent le quantificateur d'accès
  - (3) : un pointeur vers la cible { sur le schéma externe, si c'est un objet (accès) / sur la table des variables (pour un accès fictif).

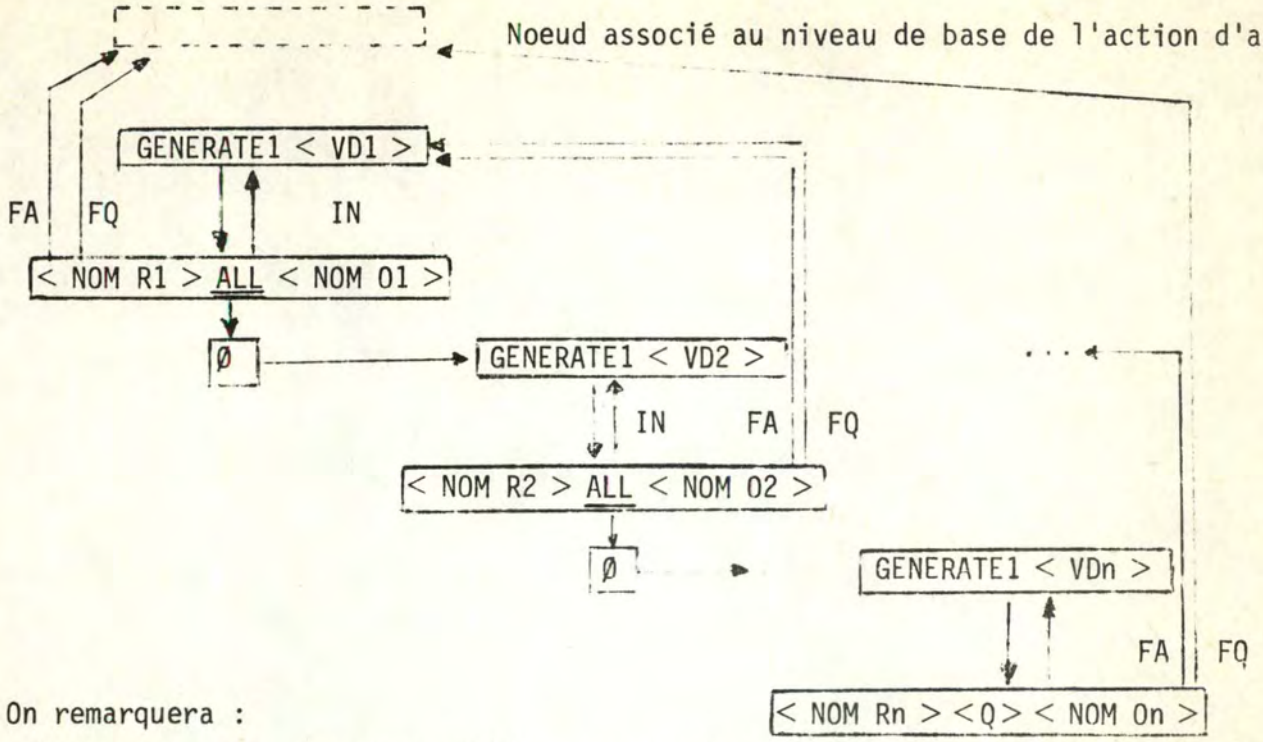
  En outre, ce noeud renferme les pointeurs suivants :
  - un père d'accès (noté FA),
  - un pointeur noté IN (*) vers un noeud GENERATE associé à une VD, ce noeud étant un ancêtre du noeud d'accès si la variable a été déclarée par une instruction GENERATE (cas [2]), ou le père direct du noeud d'accès dans le cas contraire, que la variable ait été mentionnée ou non par l'utilisateur (cas [1]).

  Les informations requises pour le quantificateur sont :
  - pour la limite inférieure de la plage de valeurs, un pointeur vers la zone des valeurs ou vers la table des variables de travail,
  - un autre pointeur pour la limite supérieure,
  - un pointeur (noté FQ) vers l'objet associé au niveau du contexte où le compteur de réalisations (associé au quantificateur) est initialisé.

  Rappelons qu'une action d'accès utilisant une relation composée est décomposée par le compilateur en autant d'actions imbriquées d'accès qu'il y a de relations simples; ces pseudo-actions intermédiaires ont "ALL" comme quantificateur et elles ne comportent pas de condition.

  La figure suivante illustre le cas général d'une relation composée de n relations et de n objets :

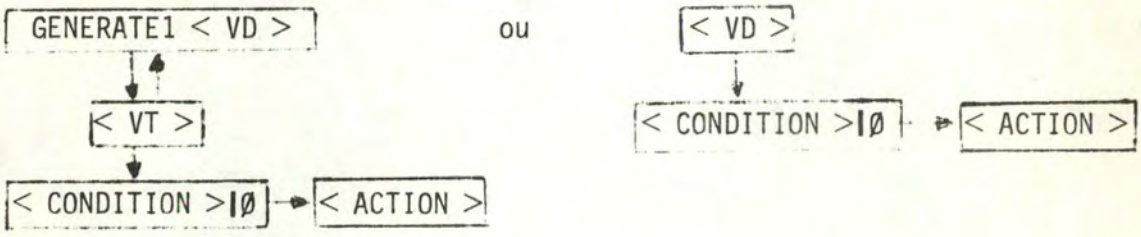
(*) En référence à la clause IN de la macro REACH du DML de l'Institut.



On remarquera :

- lorsqu'il n'y a pas de condition, on construit un noeud particulier, appelé "noeud vide";
- les pointeurs FQ des noeuds d'accès intermédiaires réfèrent le père d'accès de ces noeuds (y compris le premier, dont le père d'accès est le niveau de base), car ces accès utilisent des relations simples, alors que le pointeur FQ du dernier noeud d'accès fait référence au niveau de base de l'action d'accès.

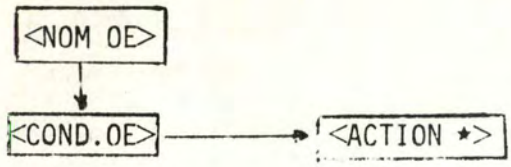
5.3.2.4. L'action à partir d'une variable



- Les noeuds de variables < VD > et < VT > contiennent un pointeur vers une table, respectivement, la table des variables de désignation et la table des variables de travail.

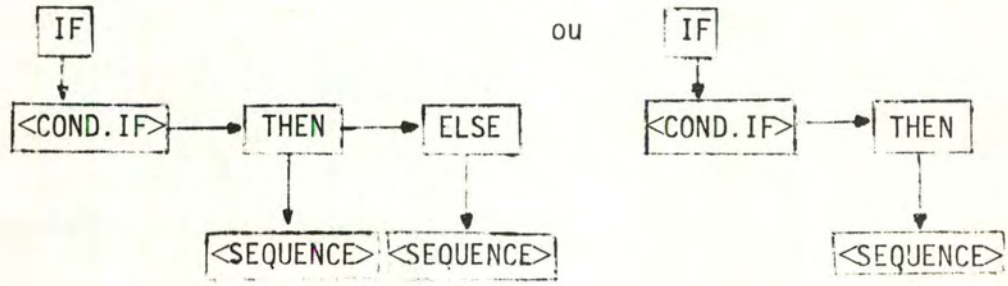
Une entrée de la table des VT fournit le nom de la variable et le type de l'objet associé (complexe ou élémentaire).

5.3.2.5. L'action à partir d'un objet élémentaire

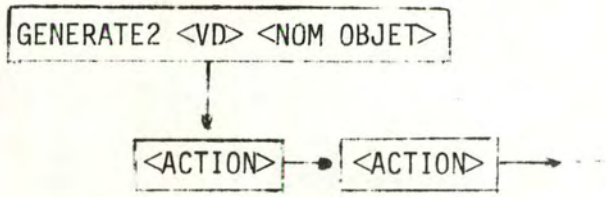


Le noeud <nom OE> contient un pointeur vers le schéma externe .

5.3.2.6. L'instruction conditionnelle

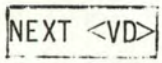


5.3.2.7. L'instruction GENERATE



Le noeud GENERATE2 contient ici, outre un pointeur vers la table des variables, un pointeur vers l'objet associé dans le schéma.

5.3.2.8. Les ordres NEXT et EXIT



Les noeuds NEXT et EXIT renferment un pointeur vers la table des VD.

- Si aucune variable n'a été mentionnée, la variable associée au noeud est la variable assignée au niveau de base du contexte de l'ordre, que cette variable ait été assignée explicitement ou qu'elle ait été générée par le compilateur.
- Si l'ordre cite une variable, il faut analyser le contexte :
  - si la variable n'a pas été déclarée explicitement par une instruction GENERATE, le pointeur du noeud fera référence à cette variable,
  - sinon, on ne peut adopter la même solution.

En effet, prenons l'exemple suivant :

```
GENERATE PERSONNE.X1
         ...
         REACH PERSONNE.X1
               ...
               NEXT X1
               ...
         END
         ...
END
```

Si le compilateur associe la variable X1 au noeud NEXT, l'ordre correspondant se comportera comme un branchement au symbole END de l'instruction GENERATE alors que l'utilisateur veut passer à la réalisation suivante de l'instruction d'accès.

Il nous faut pouvoir désigner la réalisation courante de PERSONNE autrement que par X1; c'est pourquoi le compilateur doit générer une autre VD pour ce niveau. Le noeud d'accès comportera un pointeur supplémentaire (vers la table des VD) afin de pouvoir retrouver cette variable.

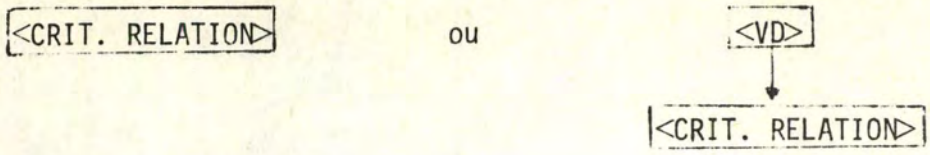
5.3.2.9. La condition

Toute condition est introduite par un noeud "expression" qui est une expression booléenne formelle de facteurs, symbolisés par des chiffres, comme par exemple ¬(1 & 2)/3 .

Un facteur est un critère d'appartenance ou un critère de relation.

A. *Le critère de relation*

- Le noeud <accès> a été vu lors de l'étude de l'action d'accès : la seule différence est qu'il n'y a pas de pointeur IN.
- Il faut signaler que l'instruction conditionnelle permet de "préfixer" un critère de relation par une variable; la représentation du critère de relation "IF" sera donc :



B. *Le critère d'appartenance*

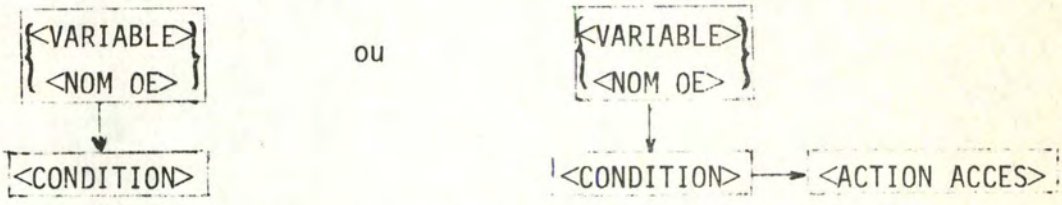
- Les opérateurs sont différenciés par l'emploi d'un code (une valeur de ce code est associée à un type de noeud).
- Le noeud <VALEUR> comporte un pointeur vers une entrée de la zone des valeurs.

Toute valeur est en fait considérée comme une liste de valeurs; une entrée de la zone des valeurs est donc associée à une liste de valeurs et les différentes listes sont séparées par un caractère spécial ("/").

Chaque valeur d'une liste comprend :

. une longueur de représentation (puisqu'elle n'est pas fixe et qu'il faut pouvoir isoler les différentes valeurs d'une liste);
. un code qui spécifie si la valeur est une constante (nombre entier, réel, string) ou une variable;
. la représentation de la constante ou le pointeur vers une table de variables.

- Le noeud <DESIGN> représente une désignation "restreinte" :



. Le noeud <VARIABLE> est un noeud <VD> ou un noeud <VT> .
. Le noeud <ACTION ACCES> figure l'ensemble des noeuds associés à une action d'accès.

5.3.3. Principes de l'analyse syntaxique

La grammaire qui engendre ADL est "context free" : elle n'est pas régulière à cause de sa structure de parenthèses.

Nous faisons allusion non seulement aux parenthèses de priorité mais aussi aux délimiteurs de critères de relation, ainsi qu'aux paires de symboles [...] et REACH ... END. Il n'existe pas d'automate fini qui accepte ce type de langage : pour analyser une chaîne d'entrée, il faut en effet compter le nombre de symboles ouvrants, puis compter le nombre de symboles fermants et s'assurer que les deux nombres sont égaux. Ceci implique qu'il faille mémoriser un nombre entier qui peut être infini, ce que ne peut faire un automate fini, dont la capacité de mémoire est inextensible.

Tout langage de ce type est accepté par un automate à pile non déterministe : ce type d'automate peut posséder plusieurs états successeurs d'un état, pour un même symbole d'entrée.

Dans notre cas, en début d'action et à la lecture du symbole "[", un tel automate aurait le choix entre l'accès normal, l'accès fictif, la création, la suppression ou la modification de relations.

Le problème est de trouver un automate déterministe équivalent; nous percevons deux approches pouvant résoudre ce problème.

- La plus simple était d'assigner à chaque action un mot-clé mnémonique, comme c'était déjà le cas de l'accès avec "REACH". Ceci aurait aussi accru la lisibilité du langage dans certains cas mais l'écriture des actions aurait été plus longue, sans compter la perte du symbolisme des boucles. Cet argument fut donc rejeté.
- L'autre était de reculer les décisions dans l'analyse de la chaîne d'entrée, tout en mémorisant les symboles précédant le symbole "décisif".

Une illustration en est fournie par l'action : [: ALL V1 ...].

C'est à la lecture du symbole V1 que l'analyseur peut décider s'il s'agit d'un accès fictif ou non.

Rappelons qu'il existe deux grandes méthodes d'analyse : la méthode "top-down" d'une part et la méthode "bottom-up" d'autre part.

- La méthode "top-down" consiste à partir du symbole distingué et à remplacer le symbole non terminal le plus à gauche par la partie droite de la production correspondante et ce jusqu'à obtention de la chaîne de caractères à analyser. Cette méthode nécessite l'usage d'une pile où sera mise la forme que doit avoir la partie de la chaîne qui n'a pas encore été lue.

  La démarche est donc essentiellement prédictive en ce sens qu'à tout stade de l'analyse et en fonction du flux de caractères déjà traités, l'analyseur prévoit que la fin du flux doit respecter une certaine forme.

- A l'antipode, la philosophie de la méthode "bottom-up" est de partir du string de caractères et de tenter de le réduire au symbole distingué. La pile d'analyse contiendra ici un sous-ensemble de symboles lus; quand l'analyseur peut y répérer la partie droite d'une production, il la réduit à sa partie gauche et ce jusqu'à ce qu'il ne reste plus que le symbole distingué au sommet de la pile.

- Ces deux méthodes impliquent donc une représentation des règles de grammaire en mémoire ainsi qu'un algorithme de recherche des parties gauches pour la première méthode et des parties droites pour la seconde.

- En outre, la définition de relations de priorité pour la méthode bottom-up peut être rendue difficile :
  - par la fonction double des parenthèses, qui sont des opérateurs de priorité et aussi des séparateurs de critères de relation;
  - par l'absence de délimiteurs de conditions, si ce n'est précisément des parenthèses.

- Enfin, la méthode d'analyse bottom-up est trop puissante pour cette grammaire qui est LL (4) au maximum, comme l'a montré l'exemple de la page précédente.

Toutes ces considérations nous ont fait rejeter l'emploi d'un automate à pile proprement dit.

Il existe cependant une autre façon d'automatiser une analyse "top down", qui nous a plu par sa simplicité : elle consiste à construire une procédure d'analyse pour chaque symbole non terminal.

Dans notre cas, certaines de ces procédures seront récursives car elles peuvent se rappeler elles-mêmes, directement ou indirectement; rappelons par exemple que le corps d'une boucle d'accès est une séquence d'instructions, qui peuvent être à nouveau des accès.

5.3.3.1. Implémentation des procédures

Si l'on veille à ce que ces procédures n'aient pas de variables locales, la zone réservée à chaque appel d'une quelconque de ces procédures ne doit donc contenir qu'une adresse de retour; l'implémentation dans un langage d'assemblage se résume à la gestion d'une pile d'adresses de retour.

Nous avons défini des macros d'assemblage à cet effet.

- La macro ENTER est la macro d'entrée dans chaque procédure récursive : elle garnit le sommet de la pile (d'adresse symbolique de début)RTNADSTK par une adresse de retour, contenue dans le registre 14.

  La première entrée de cette pile sert à sauver cette adresse tandis que la deuxième entrée contient un pointeur vers le sommet de la pile; ce sauvetage traduit la volonté d'employer le même registre 14 pour incrémenter le pointeur du sommet.

```
15900          MACRO
16000 &LAB     ENTER
16100 &LAB     EQU   *
16200          ST    14,RTNADSTK                SAVE CURRENT'PERFORM'RETURN ADDR
16300          L     14,RTNADSTK+4              LOAD STACK TOP ADDRESS
16400          LA    14,4(14)                   POINT TO NEXT STACK ENTRY
16500          ST    14,RTNADSTK+4              UPDATE STACK TOP ADDRESS
16600          MVC   0(4,14),RTNADSTK           MOVE'PERFORM'R.A.TO TOP OF STACK
16700          MEND
```

- La macro STOP est la macro de sortie de procédure : elle renvoie à l'adresse située au sommet de la pile, après avoir décrémenté le pointeur du sommet.

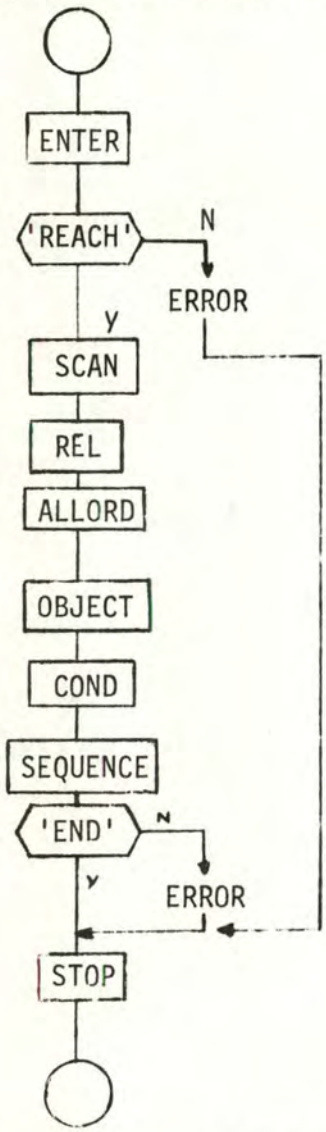
```
16800          MACRO
16900          STOP  &LAB
17000          L     14,RTNADSTK+4              LOAD STACK TOP ADDRESS
17100          S     14,=F'4'                   DECREMENT STACK TOP ADDRESS
17200          ST    14,RTNADSTK+4              UPDATE STACK TOP ADDRESS
17300          L     14,4(14)                   LOAD OLD STACK TOP ENTRY
17400          BR    14                         RETURN TO LAST'PERFORM'ADDRESS.
17500          MEND
```

5.3.3.2. Exemple d'une procédure d'analyse

Voyons par exemple la procédure qui traite l'action d'accès; dans sa version étendue, la syntaxe de cette action est :

REACH <relation> : <ordinal> <objet> <condition> <séquence> END

La procédure ACCESS s'écrit tout naturellement :



- sauver l'adresse de retour en garnissant la pile
- si le symbole courant n'est pas 'REACH', il y a erreur
- sinon, lire le symbole suivant
- appel à la procédure traitant la relation
- appel à la procédure de traitement du quantificateur
- vérifier que l'on accède bien à un objet
- traiter la condition (éventuelle)
- traiter la séquence (éventuelle) d'actions
- si le symbole courant n'est pas 'END', erreur
- sinon, brancher à l'adresse de retour du sommet de la pile

On remarquera que les procédures REL, ALLORD, COND et SEQUENCE déterminent elles-mêmes l'absence respective d'un nom de relation, d'un quantificateur, d'une condition ou d'une séquence d'actions.

La règle de lecture des symboles est que chaque procédure reçoit le premier symbole qu'elle doit traiter, dans une variable globale "NEXT SYMBOL", puis commande les lectures nécessaires à l'accomplissement de la tâche qui lui est dévolue, et enfin fournit le symbole suivant le dernier symbole qu'elle a reconnu. C'est évidemment par un appel à l'analyseur lexicographique qu'est remplie cette variable.

5.3.3.3. Traitement des erreurs

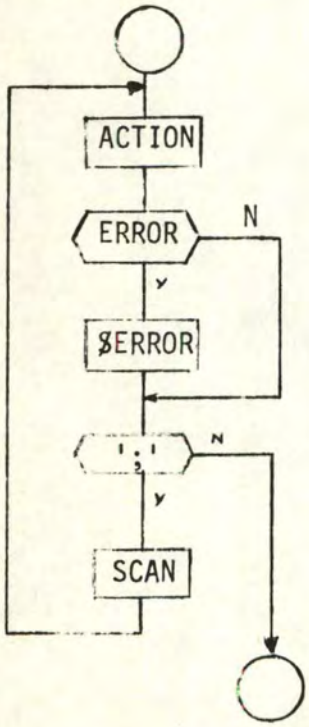
- Une des caractéristiques de cette démarche top-down est qu'un module délègue en quelque sorte une part de ses responsabilités à un autre et ainsi de suite jusqu'à ce qu'un dernier module mène son analyse à bien et renvoie le contrôle aux précédents.

Si dans la cascade d'appels, un module détecte une erreur, il ne peut que la communiquer au module appelant, qui fait de même jusqu'à ce que l'erreur ait remonté la cascade.

Cette opération est nécessaire si l'on veut définir un traitement centralisé d'erreur ainsi qu'un point de reprise cohérent de l'analyse, ce qui sous-entend une remise à l'état initial de la pile d'adresses de retour.

Un moyen simple de parvenir à cette fin est d'employer une variable globale SWERROR : un module qui découvre une erreur syntaxique positionne cette variable et abrège son analyse. Il reste alors à prévoir un test de cette variable après chaque appel de module et à court-circuiter le traitement courant si ce test se révèle négatif.

Nous avons ainsi choisi d'ignorer le reste de l'action en cours lors d'une erreur : le point de reprise est donc l'action suivante et l'erreur sera retransmise jusqu'au module traitant une séquence d'actions, dont l'organigramme est :



- traiter la première action
- donne-t-elle lieu à une erreur ?
- si oui, imprimer un message d'erreur
  lire jusqu'à la fin de l'action en cours
- sinon, le symbole suivant est-il ";" ?
- si oui, lire le symbole suivant
  et passer à l'action suivante
- sinon, la séquence est finie.

5.3.4. Construction des arborescences de programme et d'accès

Il nous faut définir les primitives suivantes :

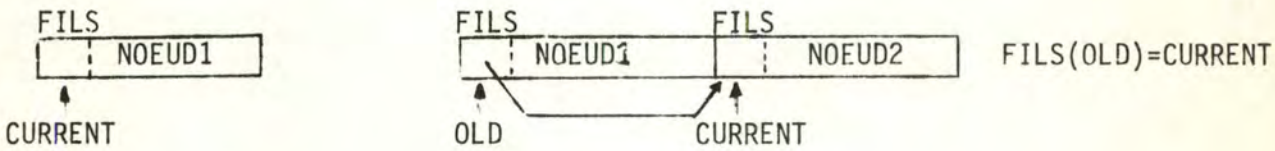
- réservation de la place occupée par un noeud
- établissement du lien "fils"
- établissement du lien "frère"
- établissement du lien "père d'accès".

5.3.4.1. Réservation de la place occupée par un noeud

Le code intermédiaire est écrit en mémoire centrale, dans une zone préalablement réservée; l'adresse initiale de cet emplacement est sauvée dans une variable CURRENT.
Cette variable joue le rôle d'un curseur se déplaçant le long de l'espace linéaire d'écriture : la primitive de réservation mettra cette variable à jour en ajoutant à son contenu la longueur du noeud courant; celle-ci est un paramètre pour la primitive étant donné la longueur variable des divers noeuds.

5.3.4.2. Etablissement du lien "fils"

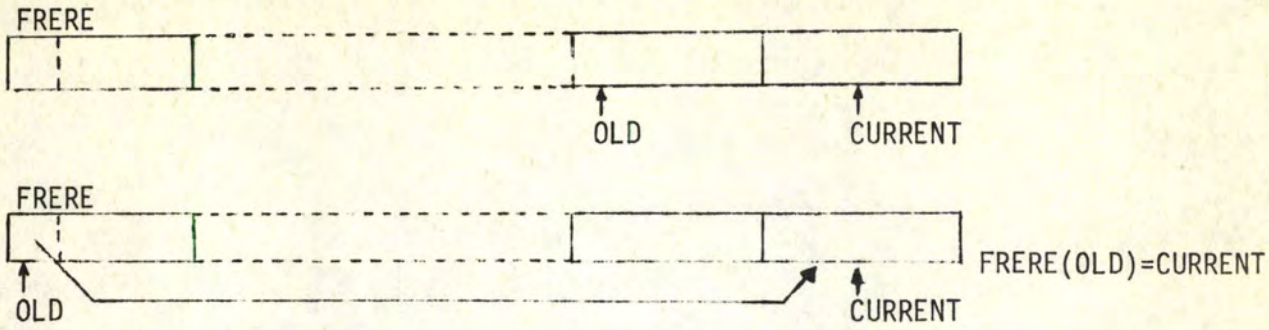
Si en outre on copie le contenu de CURRENT dans une autre variable OLD, avant de modifier la première, la primitive FILS consiste simplement à remplir le pointeur FILS du noeud pointé par OLD, par le contenu de CURRENT.



5.3.4.3. Etablissement du lien "frère"

En ce qui concerne le lien "frère", il faut remarquer que l'on est en face d'un problème du type : "dernier entré, premier traité"; le mécanisme de l'analyse haut-bas, gauche-droite est en effet tel qu'il faille traiter les descendants d'un noeud avant de se soucier de ses frères.

Les adresses des noeuds susceptibles d'avoir un frère sont retenues dans une pile CONTEXT; chaque fois que l'analyseur a fini de traiter un noeud terminal - c'est-à-dire, sans fils - la variable OLD prend la valeur du sommet de la pile (autrement dit, celle-ci pointe vers la sentinelle) et la primitive FRERE n'a plus qu'à assigner le lien FRERE du noeud OLD au noeud CURRENT, comme le montre la figure suivante.



5.3.4.4. Etablissement du lien "père d'accès"

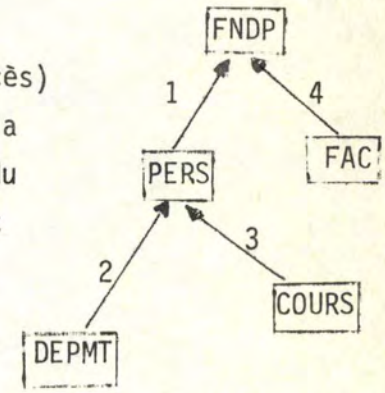
Nous avons vu que l'utilisateur du modèle de données présenté, "naviguait" dans le graphe associé à une BD en y empruntant les chemins définis par les relations d'accès. Si cette démarche était purement linéaire, il suffirait de retenir le dernier niveau atteint précédemment pour connaître le père d'accès du niveau courant.

C'est oublier d'une part, le principe de la séquence d'actions qui réajuste implicitement le niveau de base de chaque action et d'autre part, l'existence de certaines actions qui repositionnent explicitement ce niveau de base.

Exemple :

```
ENTER FNDP.XØ
      REACH : ALL PERSONNE
                  [ CHERCHEUR : DEPARTEMENT ... ]; [ TITULAIRE : COURS ... ];
                  FOR XØ [ : ALL FACULTE ... ]
      END
EXIT
```

L'arbre d'accès décrit par cet exemple est :
(la numérotation des liens exprime l'ordre des accès)
Dans la séquence, le passage de la 1ère action à la 2ème traduit la volonté du promeneur de repartir du niveau PERS précédant le niveau DEPMT qu'il venait d'accéder.
La 3ème action réajuste le niveau de base à un niveau encore supérieur, FNDP.



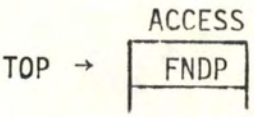
Etant donnée la portée relative de ces niveaux de contexte, nous sommes amenés à gérer une pile ACCESS : les adresses desnoeuds "niveaux d'accès" y sont empilées au début de leur portée et dépilées à la fin.

L'adresse du père d'accès d'un noeud se situera juste au dessus de l'entrée de la pile, correspondant à ce noeud, à moins d'un réajustement explicite, auquel cas il faut parcourir toute la pile.

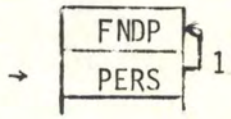
Pour illustrer ce mécanisme, convenons d'associer un noeud du CI à chaque noeud de l'arbre vu dans l'exemple et de figurer l'adresse des premiers par le nom des seconds.

Voyons les contenus successifs de la pile ACCESS au cours des opérations d'analyse.

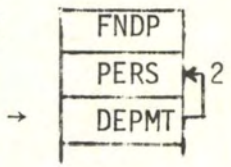
Traitement du noeud nommé FNDP



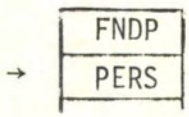
Traitement du noeud PERS : établissement du lien 1



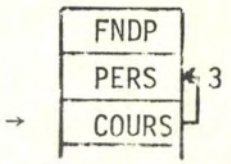
Traitement du noeud DEPMT : établissement du lien 2



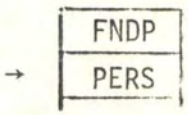
fin de portée



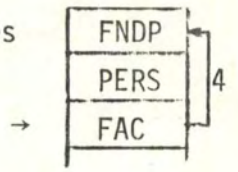
Traitement du noeud COURS : le lien 3 est établi



fin de portée



Traitement du noeud FAC : le lien 4 est établi après avoir parcouru la pile



5.4. Génération du code objet

5.4.1. Conventions

La deuxième partie de ce chapitre est constituée d'algorithmes de génération. De tels algorithmes comportent à la fois des actions à exécuter par le compilateur ADL et des actions à générer qui seront exécutées par le compilateur COBOL. Nous prendrons les conventions suivantes, en vue de séparer ces deux types d'actions.

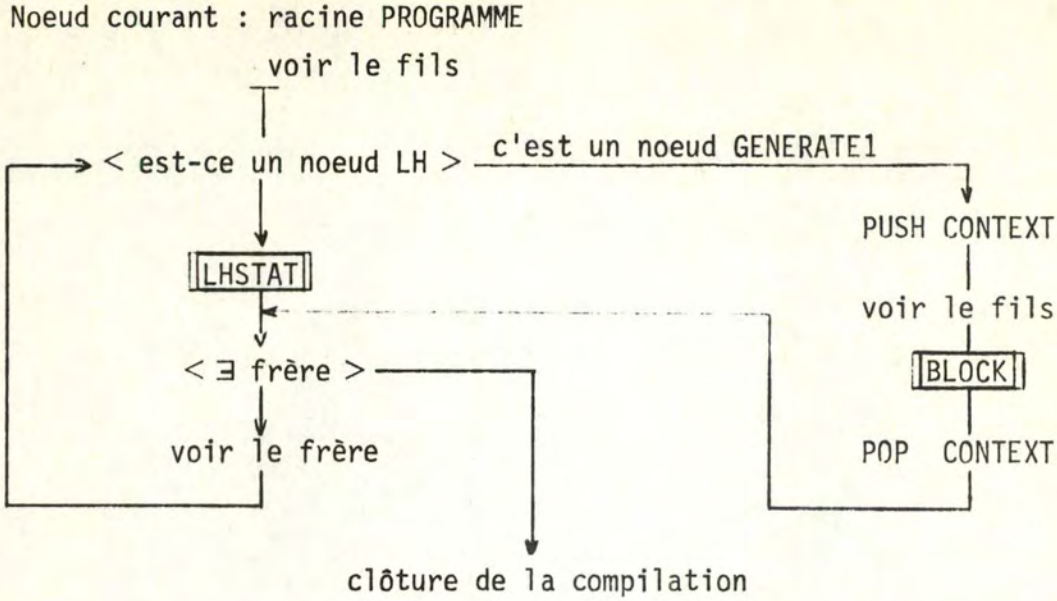
- Une proposition encadrée entre < et > constitue un test du compilateur ADL; une flèche verticale précédera les actions à effectuer en cas de réponse positive à ce test et une flèche horizontale indiquera les actions à exécuter dans le cas contraire.
- Un appel à une procédure (où à un module, une routine, un algorithme) sera figuré par une boîte encadrant le nom de cette procédure.
- Les actions générées seront directement exprimées dans le langage cible adéquat, en l'occurrence, le DML ou le COBOL; ces actions seront encadrées dans des boîtes.

5.4.2. Principe du générateur

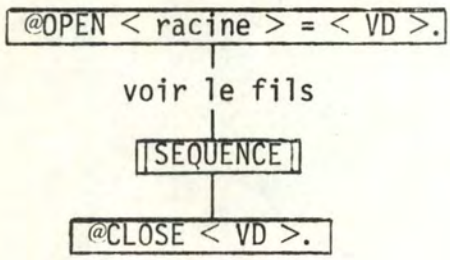
Les données du générateur sont enregistrées en mémoire centrale sous forme d'une arborescence de noeuds.
L'ordre dans lequel le générateur traite ces noeuds est sensiblement le même que celui de l'analyse : c'est l'ordre dynastique, de haut en bas et de gauche à droite. Autrement dit, lorsque le générateur a reconnu un type de noeud, il traite spécifiquement tous les descendants de ce dernier avant de passer à son frère, s'il existe, et en ayant pris soin de retenir l'adresse de ce premier noeud au sommet de la pile CONTEXT.

En outre, nous adopterons la même méthode "top-down" de génération que lors de l'analyse : les divers traitements sont confiés à des coroutines spécialisées qui peuvent s'appeler mutuellement.

La procédure directrice du générateur est décrite à la page suivante.



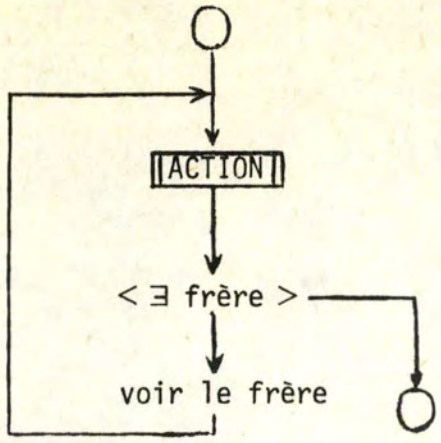
- Le module LHSTAT recopie les lignes adéquates du fichier des instructions du langage hôte FLH dans le fichier de génération FGEN.
- Le module BLOCK remplit les fonctions ci-dessous.



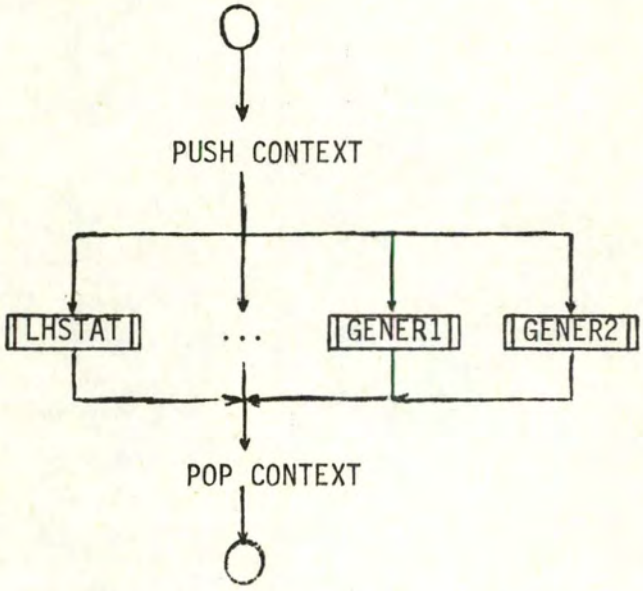
L'argument < VD > est donné par le noeud GENERATE1 précédant le noeud de la racine. Puisqu'une séquence d'actions peut comporter une action BLOCK, le module SEQUENCE peut rappeler le module BLOCK, dans notre implémentation.
Pour cette raison, il est nécessaire de sauver l'étiquette < VD > au sommet d'une pile, en début de procédure et d'aller rechercher cette étiquette, en fin de procédure. La gestion de cette pile d'étiquettes sera sous-entendue pour toutes les autres procédures qui génèrent une boucle de traitement en DML.

5.4.3. Les modules SEQUENCE et ACTION

De la même façon que dans l'analyse, le module SEQUENCE traite une séquence d'actions. Son organigramme est le suivant :

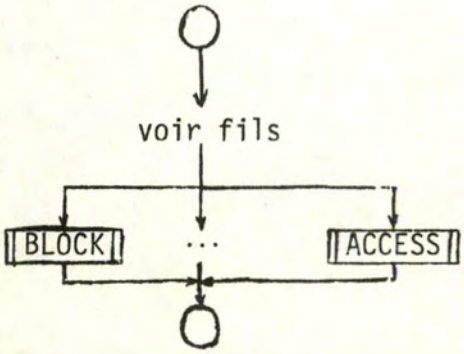


- Rappelons que chaque type de noeud est identifié par un code, dont les valeurs ont été choisies afin de dresser une table de branchement aux modules traitant ce type de noeud; c'est la procédure ACTION qui remplit cette fonction, comme suit.



Nous n'avons indiqué que les modules que nous détaillerons; il s'agit bien sûr de modules qui figurent au sommet de "hiérarchies" d'autres modules que nous expliquerons au fur et à mesure.

- Le module GENER2 correspond aux ordres GENERATE explicites du programme alors que le module GENER1 traite les noeuds du type GENERATE1 générés par l'analyseur. Ce second module base son action sur l'analyse du noeud fils du noeud GENERATE1.



5.4.4. Le module ACCESS

Cette routine verra son action influencée par le type de l'objet cible de l'accès et par le quantificateur d'accès.

- Si l'objet cible est élémentaire, il faut générer des qualifications COBOL; nous ne détaillerons pas ce cas.
- Sinon, - si le quantificateur est ##, il faut accéder à la dernière réalisation qui vérifie la condition, si elle existe.
  1. On commence par mettre à zéro une variable de travail, @CLEAR $99.
  2. Il faut générer @REACH < OC > = < VD > FROM < FA > VIA < REL > (si la relation a un nom) GET ALL.
  3. Générer le code correspondant à la condition, si elle existe; faire positionner un indicateur à 1 si elle est vérifiée et à Ø sinon; générer le test suivant :
     . si la condition est fausse, passer à la réalisation suivante
     . sinon, sauver la réalisation dans la variable $99.
  4. Générons la fin de boucle d'accès; si $99 n'est pas vide, elle contient la réalisation voulue; on y réaccède en générant la seconde forme du REACH de l'organigramme suivant.
  5. Si la condition est suivie d'une séquence d'actions, il faut enfin générer le code correspondant avant la fin de la 2° boucle.
- Sinon, le quantificateur est de la forme i#-J# (y compris ALL=1#-##).
- Générer un test : si le nombre de réalisations cibles est inférieur à la valeur de i, on fait sauter le traitement suivant (ce test n'est pas effectué dans le cas des relations qui partent de la racine).
- Après avoir généré l'ordre d'accès et le code de la condition éventuelle, il faut faire compter le nombre de réalisations qui vérifient la condition et générer les tests suivants :
  . tant que ce nombre est inférieur à i, faire passer à la réalisation suivante;
  . sinon - tant que ce nombre n'est pas supérieur à J, exécuter le code généré pour la séquence,
    - sinon, sortir de la boucle
    (si J# vaut "##", ce dernier test est superflu).
  . Terminer par la génération de l'étiquette de branchement, qui permet de sauter la boucle d'accès, le cas échéant.

Remarques générales

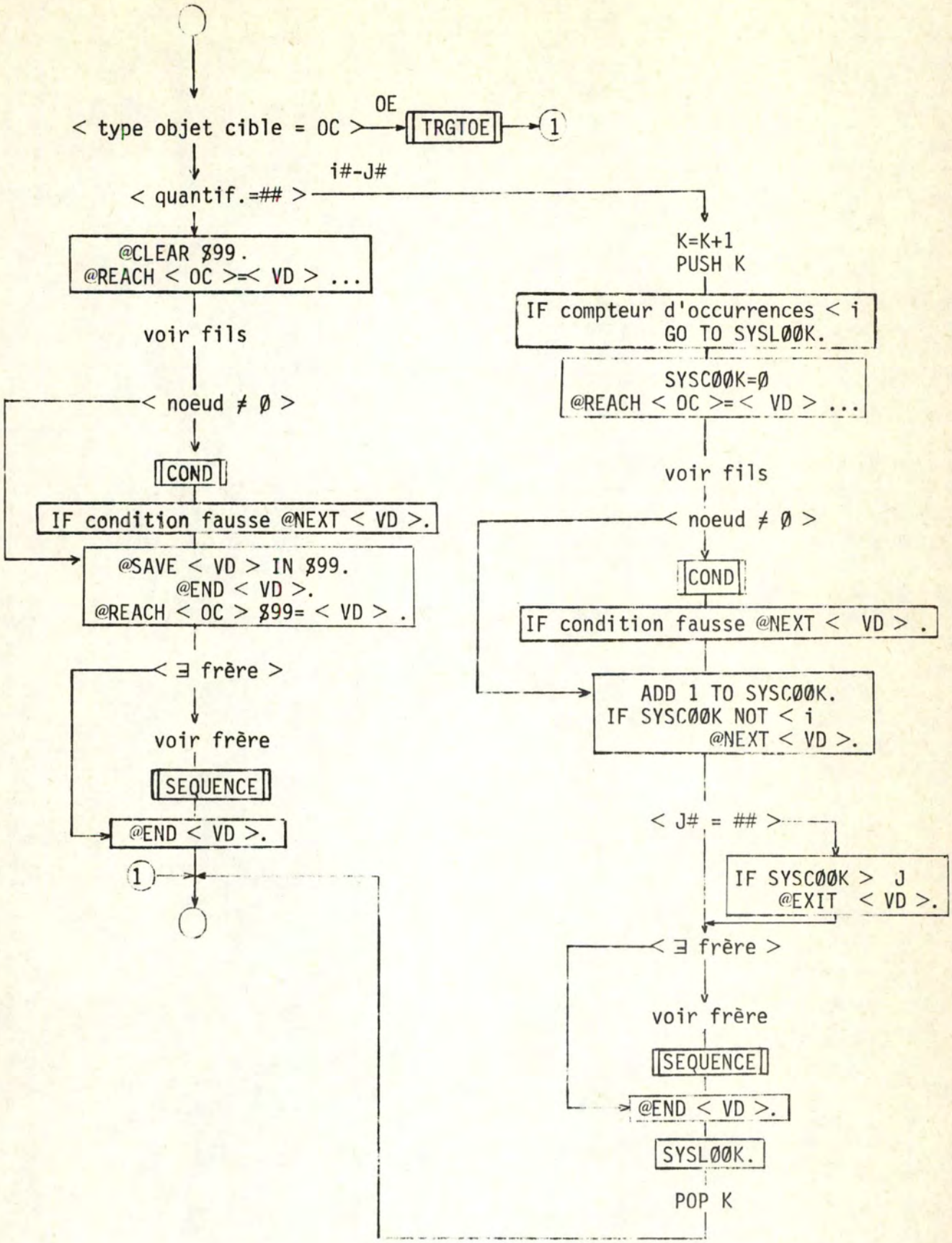
- Les variables sont traitées différemment selon qu'elles portent sur un OC ou un OE.
  A une VD d'un OC est associée une étiquette DML du même nom; à une VT contenant la référence d'un OC, est associée une VT du DML (par exemple, $99).

  Par contre, les VD d'un OE et les VT contenant la valeur d'un OE seront générées par le compilateur ADL en WORKING-STORAGE SECTION.

- Les ordres REACH générés comporteront toujours une étiquette de nom < VD > , que celle-ci ait été assignée explicitement par l'utilisateur ou qu'elle ait été générée par l'analyseur. Toujours pour simplifier la génération de ces ordres, le paramètre FROM sera employé systématiquement, suivi de l'étiquette assignée au père d'accès et le paramètre GET ALL spécifie que l'on a besoin des valeurs de tous les OE reliés à l'objet cible de l'accès.

- Les indicateurs booléens (notés SYSBØØK) et les compteurs de réalisations (SYSCØØK) sont générés en WORKING-STORAGE SECTION et sont différenciés par la juxtaposition d'un indice K alloué par le générateur. Comme une valeur de cet indice est associée à une boucle d'accès, qui peut comporter d'autres boucles, il est nécessaire de sauver les valeurs de cet indice dans une pile pour les retrouver en fin de boucle. Le même principe est applicable aux étiquettes de branchement, notées SYSLØØK.

L'organigramme du module ACCESS est repris à la page suivante.


< type objet cible = OC >
OE
TRGTOE
1
< quantif.=## >
i#-J#
@CLEAR $99.
@REACH < OC >=< VD > ...
voir fils
< noeud ≠ Ø >
COND
IF condition fausse @NEXT < VD >.
@SAVE < VD > IN $99.
@END < VD >.
@REACH < OC > $99= < VD > .
< ∃ frère >
voir frère
SEQUENCE
@END < VD >.
1
K=K+1
PUSH K
IF compteur d'occurrences < i
GO TO SYSLØØK.
SYSCØØK=Ø
@REACH < OC >=< VD > ...
voir fils
< noeud ≠ Ø >
COND
IF condition fausse @NEXT < VD > .
ADD 1 TO SYSCØØK.
IF SYSCØØK NOT < i
@NEXT < VD >.
< J# = ## >
IF SYSCØØK > J
@EXIT < VD >.
< ∃ frère >
voir frère
SEQUENCE
@END < VD >.
SYSLØØK.
POP K

5.4.5. La condition

Nous avons vu qu'une des caractéristiques du LDA était de pouvoir exprimer des désignations assez complexes en peu de mots. Il serait dommage que cet avantage formel ne s'accompagne pas d'une implémentation efficace.
Dans une brève étude [16], J.-L. HAINAUT propose deux voies d'optimisation :

- puisque le DML n'admet que des conditions sur des valeurs d'OE et que ces conditions sont directement traitées par le système de gestion de fichiers SESAM, le premier objectif sera de donner au compilateur DML le maximum de conditions à évaluer;
- d'autre part, l'évaluation des autres conditions à vérifier explicitement pour chaque réalisation fournie par le système de gestion, sera abandonnée dès que l'expression booléenne prend la valeur "vrai" ou "faux".

Ces deux fonctions seront remplies respectivement par l'algorithme de factorisation et par l'algorithme de génération et d'évaluation.

Décrivons le module COND, qui traite un noeud de condition.

- Au départ, un indicateur booléen est attaché à toute la condition; il est généré en WORKING-STORAGE SECTION et sa valeur initiale est Ø ("faux").
- L'expression de la condition est ensuite factorisée; l'algorithme de factorisation transforme toute expression booléenne B en un produit de deux expressions (EP) & (ENP) tel que :
  - EP reprenne le maximum d'éléments de B et obéisse à la structure des conditions directement exprimables en DML,
  - ENP reprenne le minimum d'éléments de B et représente les conditions à vérifier explicitement.

  Il est alors possible de générer la boucle d'accès suivante, si EP n'est pas vide.

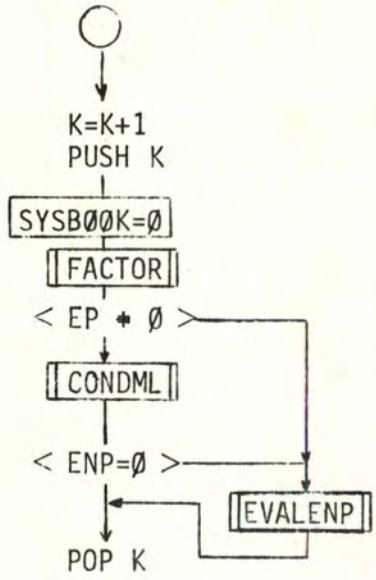
```
@REACH < OC >=< VD > ... IF (EP).
       < génération de ENP >
       IF ENP fausse THEN
                          @NEXT < VD >.
       < séquence >
@END < VD >.
```

- Si la condition EP est fausse, la condition toute entière est fausse

puisqu'un de ses facteurs est faux et le corps de la boucle ne sera pas exécuté pour la réalisation courante.

- Si ENP n'est pas vide, il faut générer cette condition et l'évaluer; le résultat de cette évaluation sera affecté à toute la condition.

L'organigramme du module COND résume ces différentes fonctions.



5.4.5.1. L'algorithme de factorisation

Pour simplifier le raisonnement donnons-nous quelques définitions.

- Un critère simple est pointé s'il est un critère de relation portant sur la valeur d'un OE.
- Une somme est une expression pointée si tous ses termes sont pointés.
- Un produit est une expression pointée si l'un de ses facteurs est pointé.

Par exemple, si l'on convient de représenter par une lettre, une condition complémentée ou non, on peut dire :

$\dot{a}/b$ &$(\dot{c}/d$ & $(\dot{e}$ & $f/\dot{g}))$ est une expression pointée;

a & $(b/\dot{c}$ & $\dot{d})$ ne l'est pas.

Rappelons qu'en DML, une expression générale de conditions doit être un produit de facteurs qui sont des sommes de conditions.

Dès lors, l'algorithme de factorisation a pour but de transformer toute expression en un produit (EP) & (ENP) tel que :

- EP est vide ou est un produit de sommes de critères tous pointés.
- ENP est vide ou est un produit de sommes de critères qui ne sont pas tous pointés.

Si y, EP, ENP sont des variables pouvant contenir des expressions, on définit les primitives suivantes :

- EP ← EP . y est la concaténation sous forme de produit des expressions de EP et de y, éventuellement parenthésées, et le rangement du résultat dans EP.
  Ainsi, si EP contient "a & b" et y, "c/d", l'opération range "a & b & (c/d)" dans EP.
- ENP ← ENP x y signifie :
  . si ENP est vide, le contenu de y est copié dans ENP;
  . sinon, le produit des expressions de ENP et de y est développé et rangé dans ENP.

Si ENP contient a & b et y contient c/d, cette opération range a & b & c/a & b & d dans ENP.

D'autre part, on utilise la procédure NORMAL qui transforme une expression en une somme de produits de conditions et qui indique si cette expression est pointée ou non.

Nous pouvons passer à l'algorithme proprement dit.

Toute expression est vue comme un produit d'un ou plusieurs facteurs.

On considère successivement chacun de ces facteurs pour lequel on effectue ce qui suit :

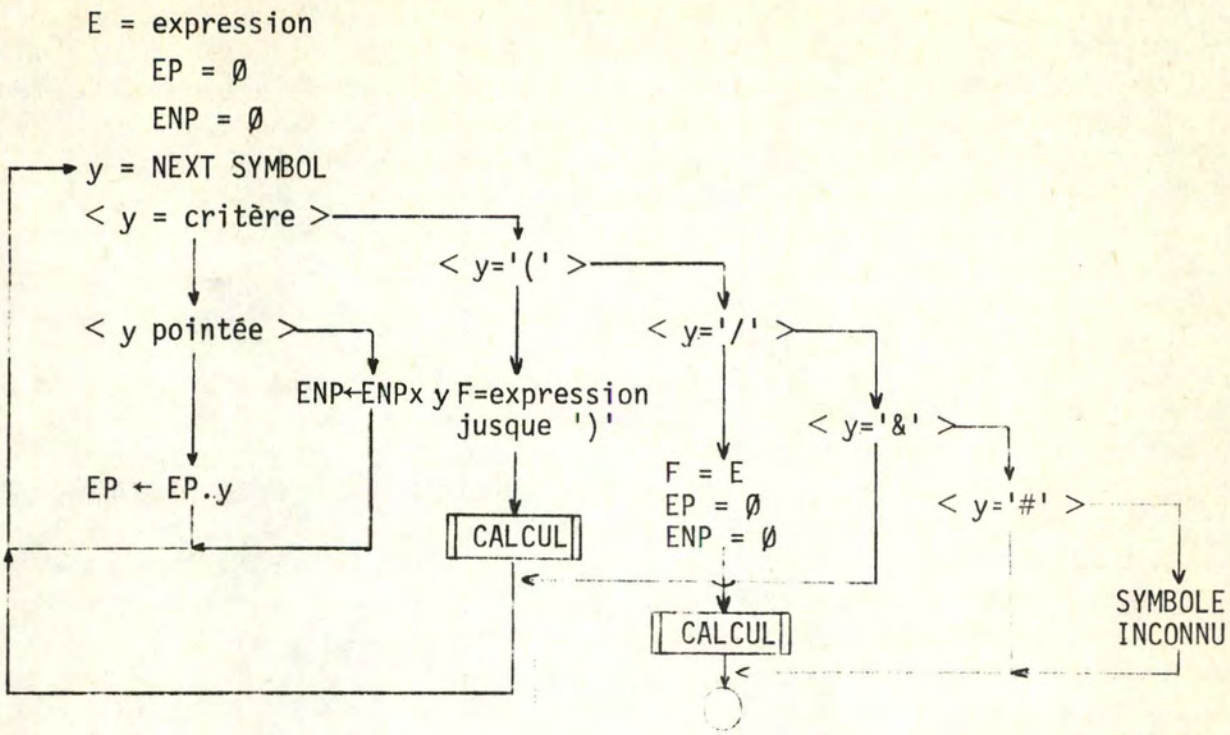
- ranger ce facteur dans la variable F
- transformer F en F' par NORMAL
- Si F' n'est pas pointée, ENP ← ENP x F'
- sinon, . choisir dans chaque terme de F' une condition pointée Ci
  . construire l'expression f = C1/C2/.../Ci/.../Cn
  . EP ← EP.f
  . si f est différent de F', ENP ← ENP x F'
  . sinon, ENP reste vide.

Remarque : si une expression est pointée, alors l'algorithme en extraira un facteur EP non vide.

Exemples de transformation

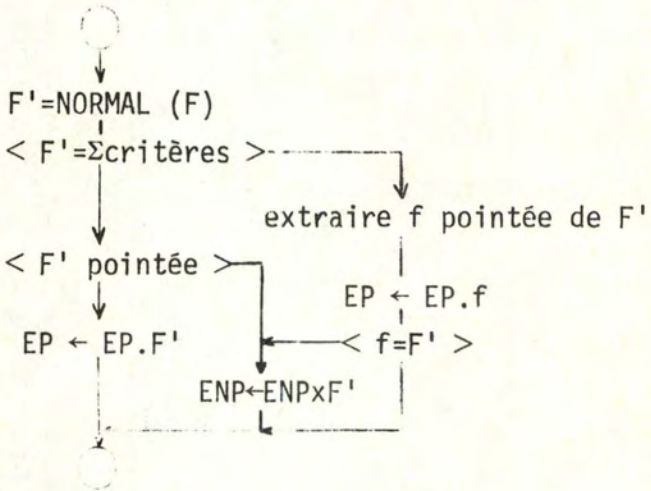
1. $\dot{a}/b \;\&\; \dot{c}$	⇒	EP = $\dot{a}/\dot{c}$	ENP = $\dot{a}/b \;\&\; \dot{c}$
2. $\dot{a}/b$	⇒	EP = Ø	ENP = $\dot{a}/b$
3. $\dot{a} \;\&\; (\dot{b}/\dot{c}) \;\&\; \dot{d}$	⇒	EP = $\dot{a} \;\&\; (\dot{b}/\dot{c}) \;\&\; \dot{d}$	ENP = Ø

Voici l'organigramme du module FACTOR.



Pour accélérer l'analyse, on envisage les cas particuliers où E est un simple critère ou un produit de critères.

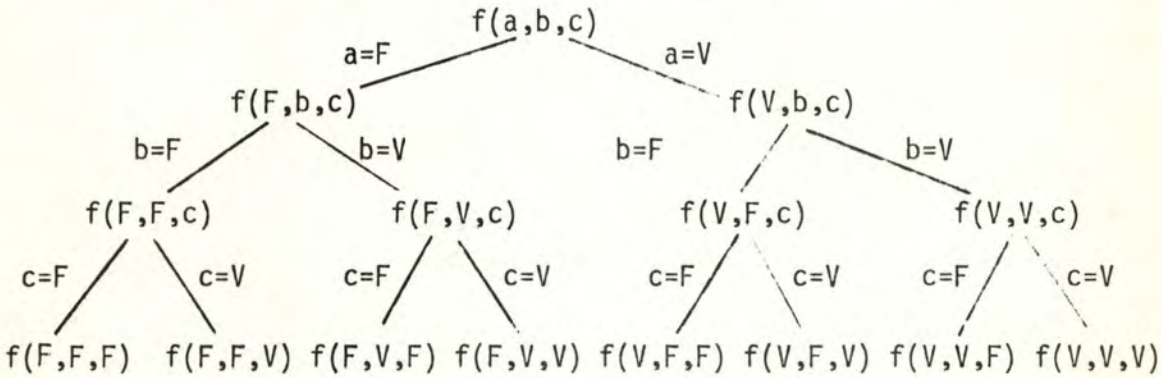
Les autres cas sont traités par la procédure CALCUL, qui peut se présenter comme suit :



Cette fois, on prévoit le cas particulier où E se réduit à une somme de critères.

5.4.5.2. L'algorithme de génération et d'évaluation d'une expression booléenne

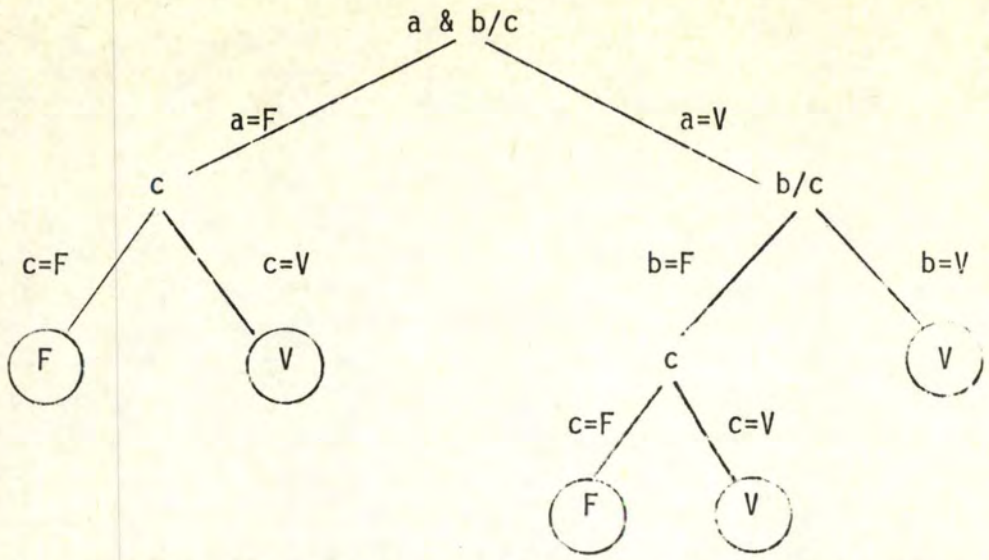
L'expression à générer est celle que l'algorithme de factorisation a rangée dans ENP et qui se présente sous forme d'une somme de produits.
Prenons le cas d'une expression de 3 conditions, que nous symbolisons par f(a, b, c). Puisque chacune des conditions peut prendre la valeur F ou V, il y a 8 possibilités dans l'évaluation de cette expression. On peut schématiser cette évaluation par une arborescence d'expressions.



Il serait avantageux de ne pas parcourir toute cette arborescence en arrêtant l'évaluation dès que possible.
L'algorithme qui remplit cette fonction est le suivant (nous l'appellerons algorithme de réduction).
On part d'une expression qu'on a copiée dans une variable et on suppose que l'on a isolé une condition de cette expression.

- Si la condition est fausse, on supprime dans la variable tous les termes dans lesquels la condition apparaît. Si après cette mise à jour, la variable est vide, l'expression est reconnue fausse et on peut faire arrêter l'évaluation. Sinon, il faudra poursuivre l'évaluation en choisissant une autre condition.
- Si la condition est vraie, on en élimine toutes les occurrences dans la variable. Si cette mise à jour fait disparaître un terme, l'expression est tenue pour vraie et on peut faire arrêter l'évaluation. Sinon, il faudra la poursuivre.

Prenons le cas particulier de l'expression f=a & b/c; l'algorithme de réduction peut être employé comme suit.



Le problème est d'appliquer l'algorithme de réduction à chaque expression associée à un noeud de l'arborescence.

L'algorithme d'évaluation et de génération peut s'énoncer comme suit :

1. Il faut choisir une condition de cette expression, la générer, lui supposer d'abord la valeur fausse et lui appliquer l'algorithme de réduction.
2. Si cet algorithme ne délivre pas de verdict final, il faut recommencer la 1° étape avec une autre condition.
3. Dans le cas contraire, il faudra retrouver l'expression et supposer maintenant que la condition choisie est vraie, puis appliquer l'algorithme de réduction et reprendre la 2° étape.

Nous sommes amenés à gérer une pile dont une entrée est composée de 4 arguments :

EXP(i) est une expression booléenne, ou Ø (faux), ou 1 (vrai);
VAR(i) est le numéro de la condition que l'on va évaluer;
VAL(i) est la valeur (Ø ou 1) supposée de la condition VAR(i);
LAB(i) est l'indice K de l'étiquette du paragraphe traitant le cas où la condition VAR(i) a la valeur "vraie".

L'interprétation de la pile est la suivante : EXP(i+1) sera l'expression EXP(i) réduite dans le cas où la condition VAR(i) prend la valeur VAL(i).

La génération sera structurée comme suit :

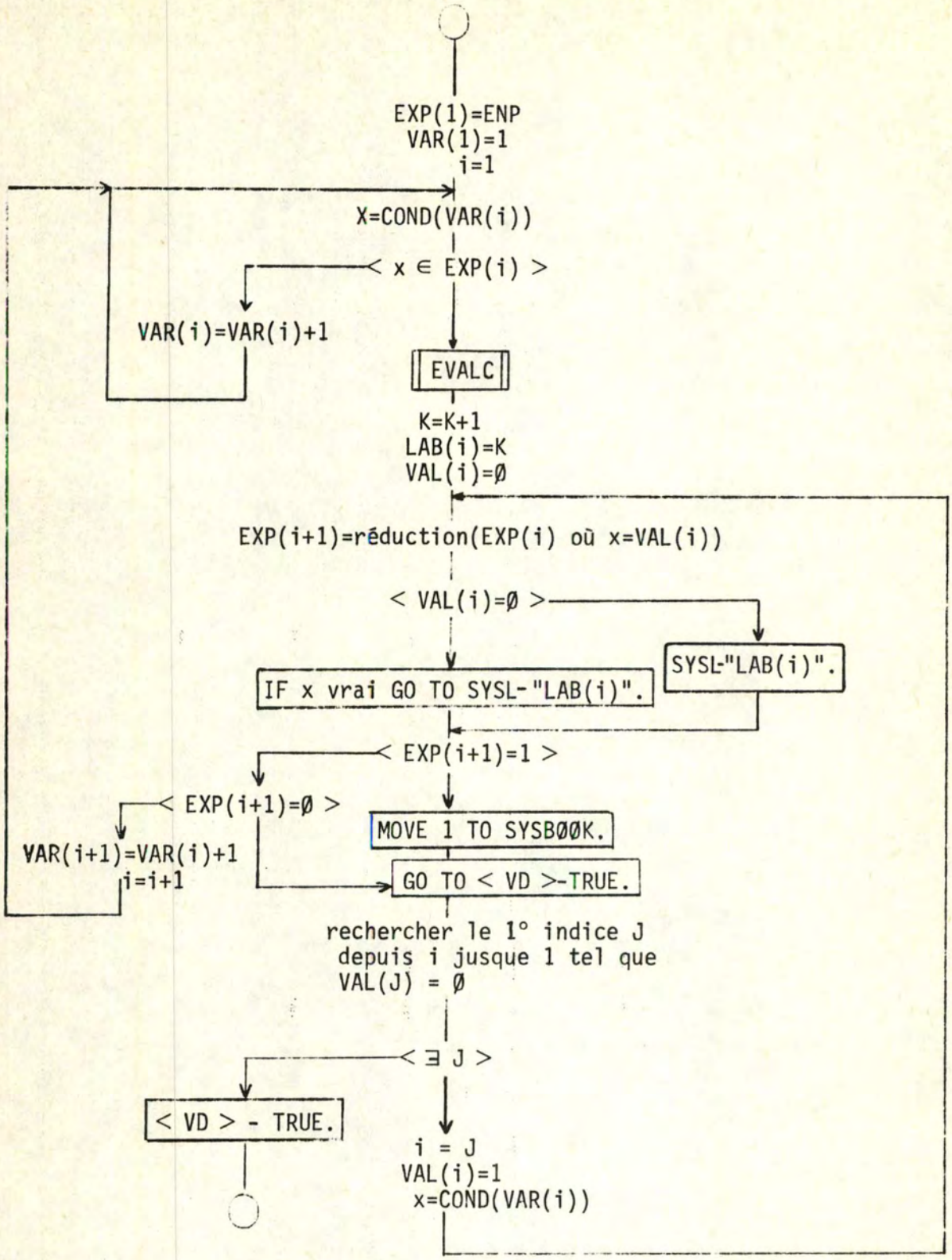
- Pour le critère VAR(i), on aura :

```
évaluer la condition VAR(i)
IF vraie GO TO SYSL-"LAB(i)".
        < évaluation des cas où VAR(i) est fausse >
SYSL-"LAB(i)".
        < évaluation des cas où VAR(i) est vraie >
```

- Pour toute la condition, on aura :

```
@REACH < OC > = < VD > ... IF(EP).
        évaluer la 1° condition
        IF vraie GO TO SYSL-"LAB(1)".
                < évaluation des cas où la 1° condition est fausse >
        SYSL-"LAB(1)".
                < évaluation des cas où la 1° condition est vraie >
        < VD > - TRUE.
                IF condition fausse @NEXT < VD > .
                < séquence >
@END < VD >.
```

Nous pouvons passer à l'organigramme de cet algorithme, appelé EVALENP. Signalons encore que le vecteur COND reprend la liste des conditions qui constituent l'expression ENP et que la routine EVALC sert à générer la condition contenue dans la variable x.


EXP(1)=ENP
VAR(1)=1
i=1
X=COND(VAR(i))
< x ∈ EXP(i) >
VAR(i)=VAR(i)+1
EVALC
K=K+1
LAB(i)=K
VAL(i)=Ø
EXP(i+1)=réduction(EXP(i) où x=VAL(i))
< VAL(i)=Ø >
IF x vrai GO TO SYSL-"LAB(i)".
SYSL-"LAB(i)".
< EXP(i+1)=1 >
< EXP(i+1)=Ø >
VAR(i+1)=VAR(i)+1
i=i+1
MOVE 1 TO SYSBØØK.
GO TO < VD >-TRUE.
rechercher le 1° indice J
depuis i jusque 1 tel que
VAL(J) = Ø
< ∃ J >
< VD > - TRUE.
i = J
VAL(i)=1
x=COND(VAR(i))

- Cet algorithme suppose l'existence d'une fonction de génération d'une condition, EVALC; si celle-ci concerne un OE, cette fonction sera assez simple ; si par contre, la condition concerne un OC et si celui-ci est également soumis à une expression de conditions, la fonction devra faire appel à l'algorithme ci-dessus. Le programme correspondant sera donc récursif.
- Le principe de cet algorithme admis, le moyen principal d'accélérer l'évaluation réside dans la stratégie du choix de la condition à évaluer.

a. Les conditions sur OE seront évaluées en premier lieu.

b. Si l'on accepte de consulter le graphe de la BD, on peut évaluer les conditions portant sur des relations dans l'ordre des valeurs croissantes de leur caractéristique J. Par exemple, "les facultés dont un professeur s'appelle x et dont le doyen est y" seront obtenues plus rapidement si l'on évalue d'abord la seconde condition.

5.4.5.3. Exemple

Soit l'instruction d'accès suivante :

```
[ : PERSONNE((: NOM='Z')/(PROFESSEUR : FACULTE(: NOM='DROIT'))
                    & (: ADRESSE(: VILLE='NTB'))) < séquence >]
```

La condition présentée a la forme $E=\dot{a}/b \;\&\; \dot{c}$

L'algorithme de factorisation décompose E en $\begin{cases} EP=\dot{a}/\dot{c} \\ ENP=\dot{a}/b \;\&\; \dot{c} \end{cases}$

L'algorithme de génération et d'évaluation part de $EXP=\dot{a}/b \;\&\; \dot{c}$: il choisira d'abord a puis c puis b; le vecteur COND contient

a
c
b

Voici les contenus successifs de la pile,
(on suppose que K vaut 21 au départ)

	EXP	VAR	VAL	LAB	
i=1	a/b & c	1	Ø	22	on suppose que a est faux
2	b & c	3	Ø	23	on suppose que c est faux
3	Ø				

1	a/b & c	1	Ø	22	
2	b & c	3	1	23	supposons maintenant que c soit vrai
3	b	2	Ø	24	on suppose que b est faux
4	Ø				

1	a/b & c	1	Ø	22
2	b & c	3	1	23
3	b	2	1	24
4	1			

supposons maintenant que b soit vrai

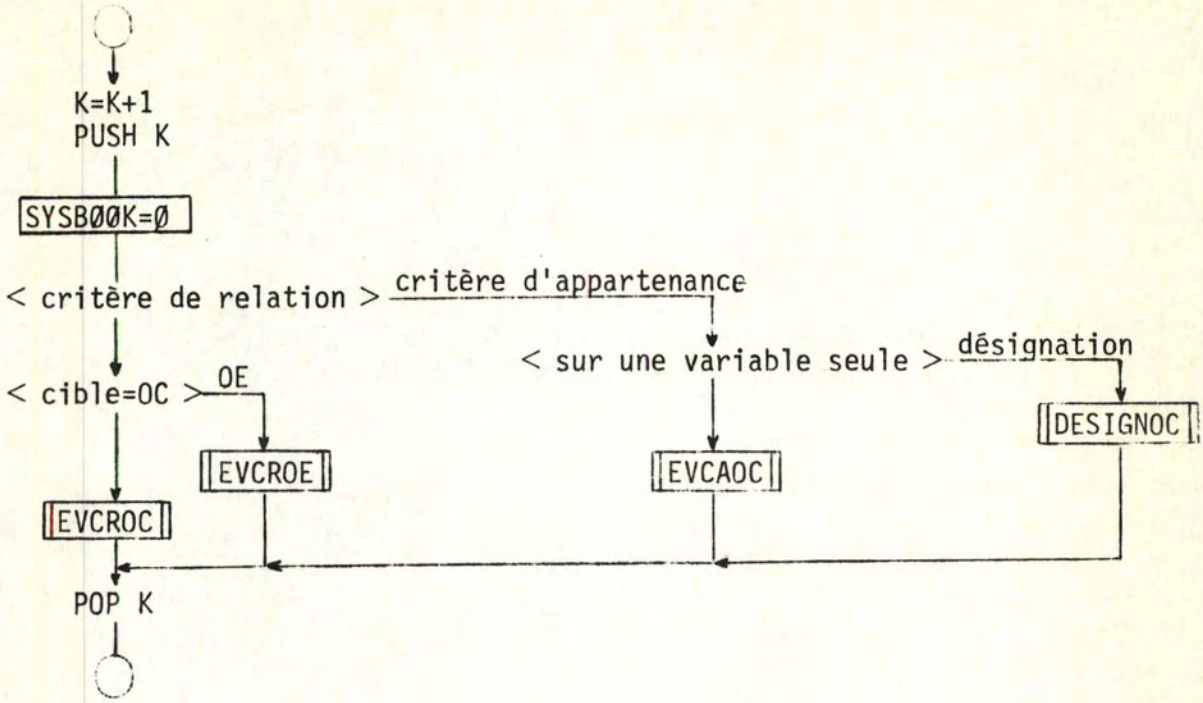
1	a/b & c	1	1	22
2	1			

supposons enfin que a soit vrai

L'algorithme génère le programme suivant :

```
   générer a
   IF a vrai GO TO SYSL22.
   générer c
   IF c vrai GO TO SYSL23.
   GO TO P1-TRUE.
SYSL23.
   générer b
   IF b vrai GO TO SYSL24.
   GO TO P1-TRUE.
SYSL24.
   MOVE 1 TO SYSBØØK
   GO TO P1-TRUE.
SYSL22.
   MOVE 1 TO SYSBØØK
   GO TO P1-TRUE.
P1-TRUE.
```

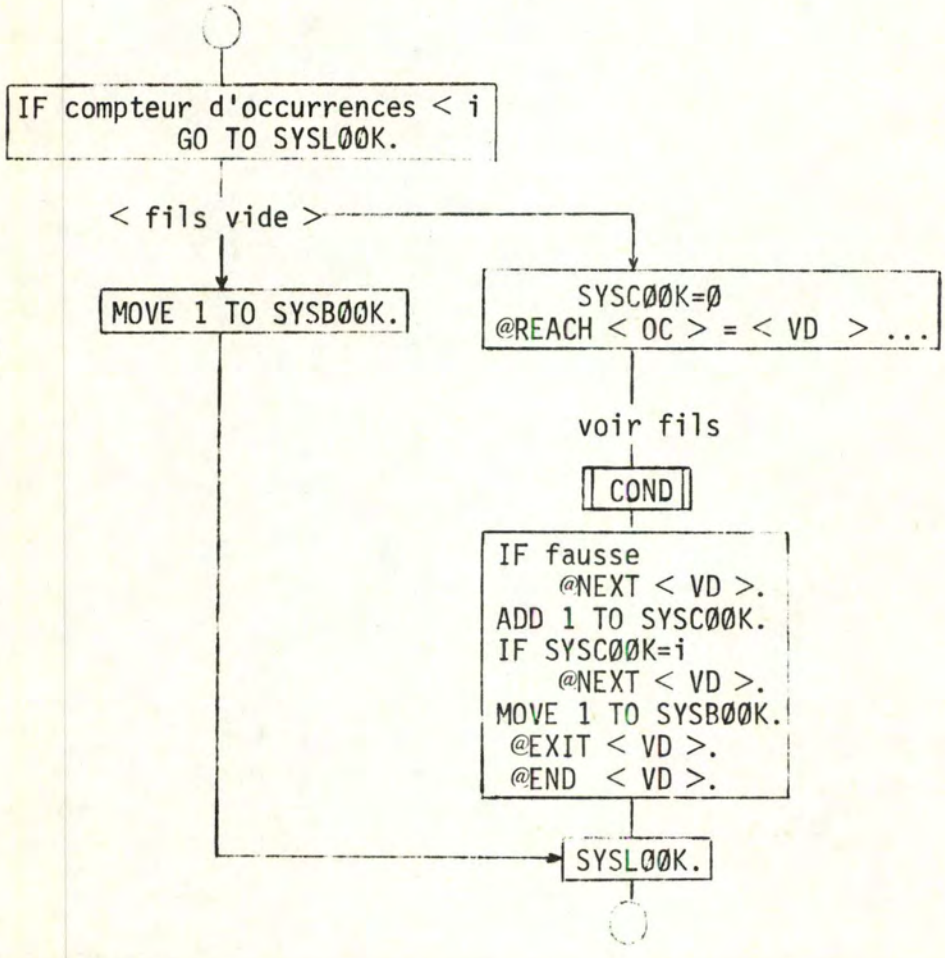
5.4.5.4. La routine de génération d'une condition EVALC



5.4.5.5. La génération du critère de relation sur OC (EVCROC)

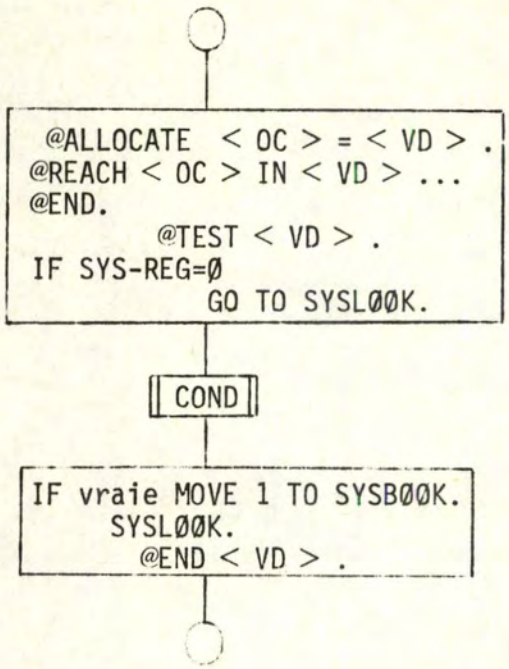
1° cas : le quantificateur est de la forme I-

Le principe est de générer une boucle d'accès si ce critère est suivi d'autres conditions, puis d'y compter les réalisations qui vérifient ces conditions subordonnées et d'en sortir après i succès.



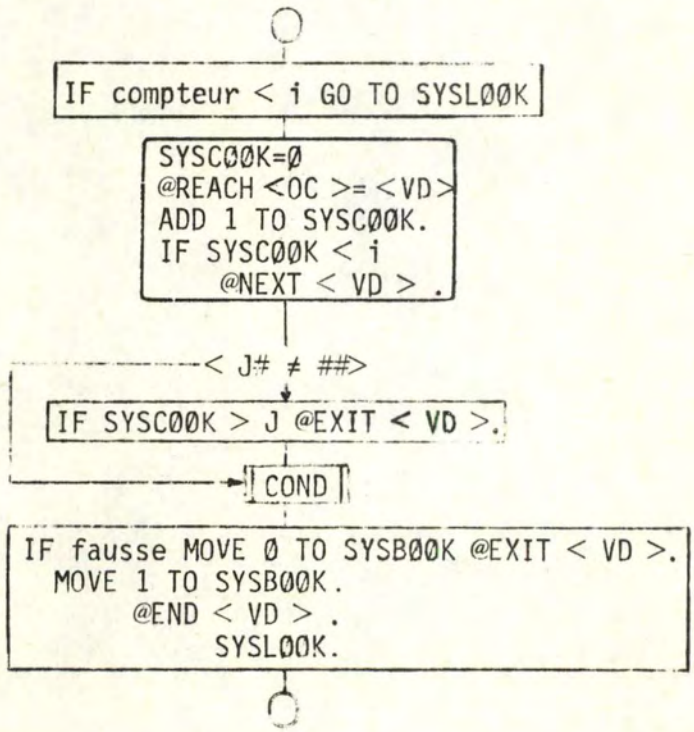
2° cas : le quantificateur est ##

On génère une boucle d'accès que l'on fait exécuter normalement; à la fin de cette exécution, on possède la dernière réalisation et il reste à vérifier la condition.



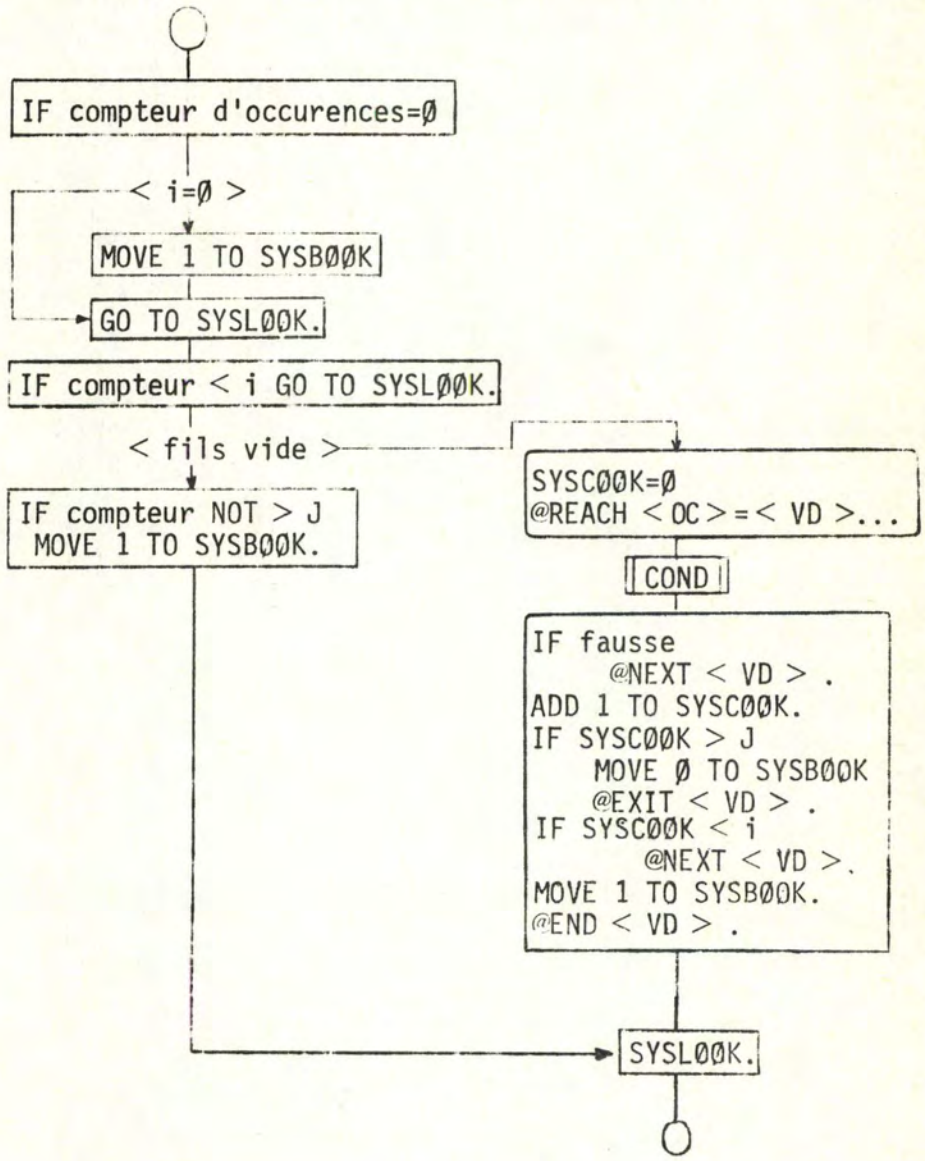
3° cas : le quantificateur est du type I#-J#

On accède à la i° réalisation de la boucle en comptant les réalisations et à partir de là, on examine les réalisations suivantes, jusqu'à la J° : si elles vérifient toutes la condition subordonnée, la condition entière est vraie; autrement dit, dès qu'une de ces réalisations ne convient pas, la condition est jugée fausse et l'on sort de la boucle.



4° cas : le quantificateur est du type I-J

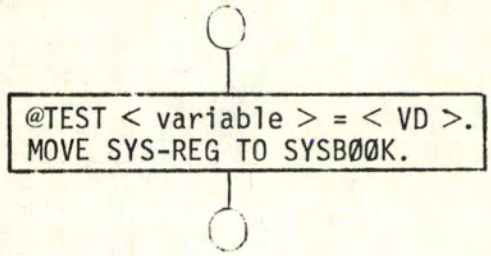
Rappelons que ce cas inclut le cas J+ qui est considéré comme étant du type Ø-J.
S'il n'y a pas de réalisation cible et si i vaut Ø, la condition est dite vraie; si i n'est pas nul, elle est fausse.
Si le nombre de réalisations cibles est inférieur à i, la condition est encore fausse; sinon, on va voir s'il y a une condition subordonnée : s'il n'y en a pas, il faut que le nombre de réalisations cibles ne soit pas supérieur à J, pour que la condition soit vraie.
S'il y a une condition subordonnée, on compte les réalisations cibles qui la vérifient; s'il y en a plus de J, la condition est fausse et l'on sort de la boucle; sinon, il faut encore s'assurer que ce nombre n'est pas inférieur à i.



5.4.5.6. La génération du critère d'appartenance sur OE.

1° cas : le second membre est une variable seule

La routine EVCAOC se borne à tester l'égalité de cette variable et de la variable de désignation associée à la boucle d'accès à l'OC.
La condition est fausse si ces deux variables ne sont pas égales ou encore si la première est vide.

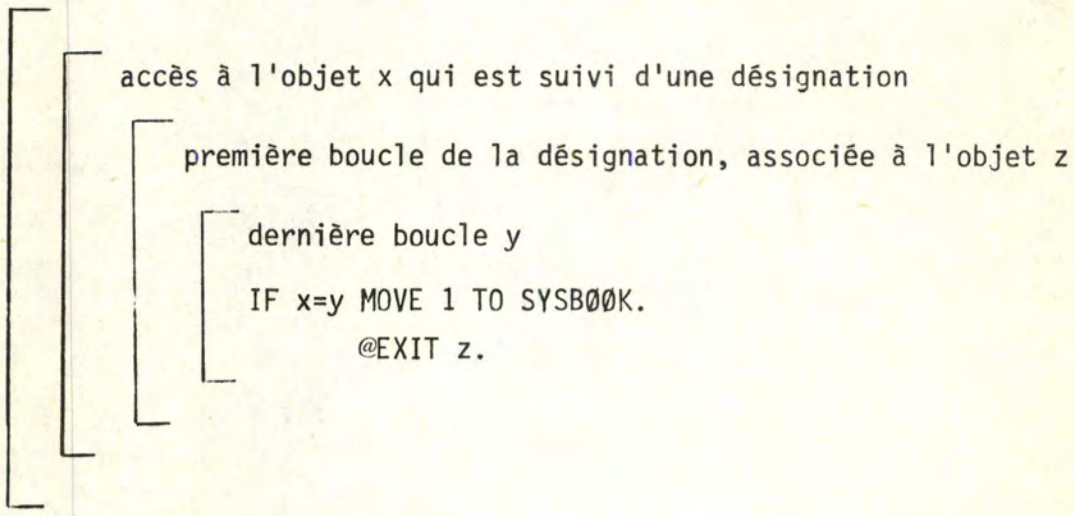


2° cas : le second membre est une désignation

Une désignation peut commencer par une variable. Si la variable est soumise à une condition, elle doit la satisfaire : si ce n'est pas le cas, la condition entière est déclarée fausse et il ne faut pas exécuter le code généré pour la désignation par accès.
La désignation peut aussi débuter par un OE; la condition qui porte sur cet OE sera reprise dans la clause IF de la première boucle d'accès de la désignation par accès.

La désignation peut enfin se faire à partir d'une racine; il faut générer l'ordre d'accès à la BD correspondante avant de traiter la désignation par accès, puis il faut fermer le contexte associé à la BD.

La désignation par accès pose un problème particulier, que nous illustrons en représentant une boucle d'accès par un crochet.

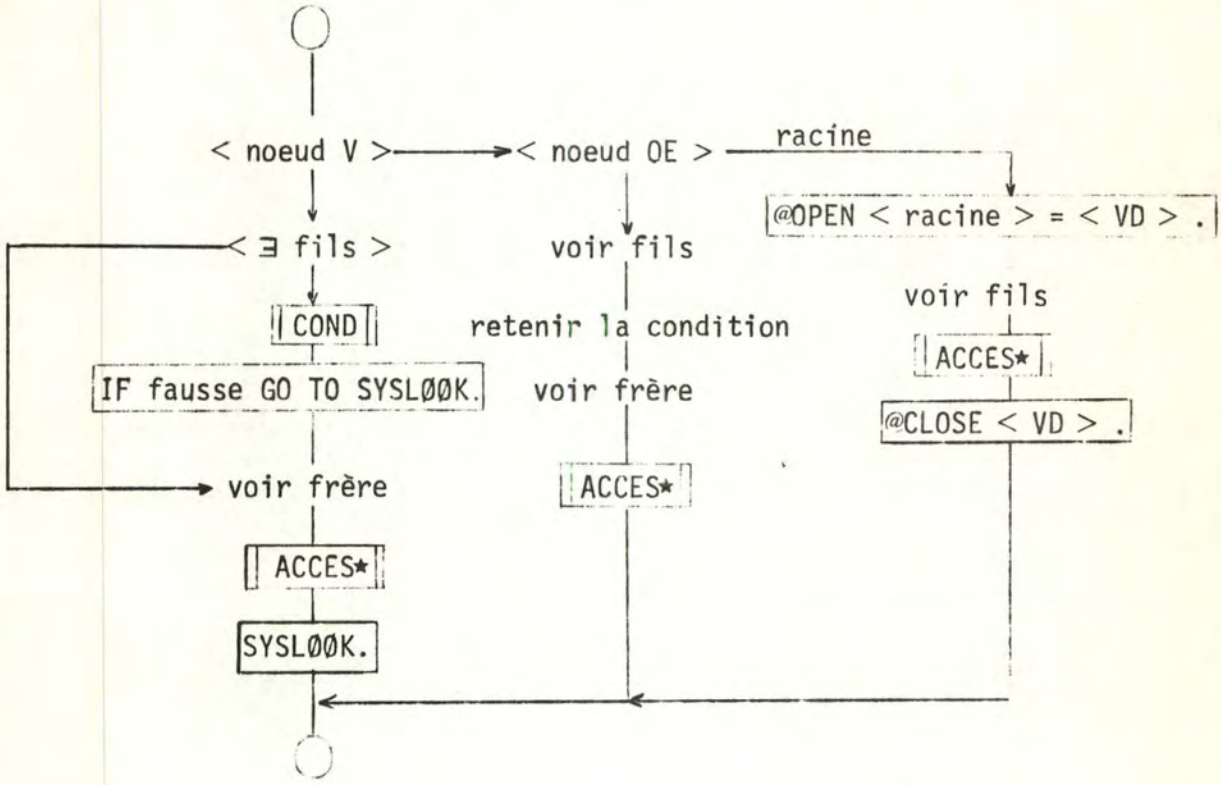


C'est dans le contexte de la dernière boucle d'une désignation par accès qu'il faut insérer le test d'égalité des réalisations origine et cible de la désignation; on ne peut donc reprendre telle quelle la procédure récursive ACCESS.
La solution adoptée est de réécrire une autre procédure ACCES* où la récursivité est "dégénérée" en une boucle. Cette solution est faisable pour les raisons suivantes :

- dans l'accès d'une désignation, la séquence d'actions ne peut être qu'un accès ou une action à partir d'une variable, dont l'action est un accès et qui est dès lors générée comme un accès (le paramètre FROM sera suivi de la variable);
- dans la procédure ACCESS, les seules opérations à effectuer après l'appel de la procédure SEQUENCE étaient des générations de fins de boucle et d'étiquettes, qui sont sauvées dans des piles.

Il suffit donc de répéter cet épilogue d'opérations autant de fois que l'on a généré de boucles d'accès et on peut alors distinguer la dernière boucle pour y insérer le test adéquat.

Voici l'organigramme du module DESIGNOC.



5.4.5.7. La génération du critère sur OE

La difficulté réside dans le fait que le DML traite différemment les OE et les OC alors qu'ADL permet l'emploi de quantificateurs dans les 2 cas. L'approche la plus "naturelle" consiste à travailler directement sur les valeurs des OE visés, qui sont disponibles au sommet de la pile de mémoire garnie par les ordres REACH du DML. Dans un critère de relation, le nombre de valeurs qui vérifient une condition subordonnée est fixé par la répétition d'une opération d'incrémentation; dans cette approche, la procédure EVCROE doit traiter toutes les conditions subordonnées jusqu'au critère d'appartenance, et le problème est pratiquement insurmontable, du fait qu'on ne peut le résoudre par "étapes".

Dans le cas des OC, la procédure EVCROC répétait un tel test par l'exécution d'une boucle d'accès et les critères de relation emboîtés étaient traités par récursivité.

Il semble qu'une meilleure approche serait de structurer la génération relative aux OE de la même manière que pour les OC, et en particulier de générer des boucles d'acquisition des différentes réalisations d'un OE cible; une technique simple et systématique serait de simuler cet accès par un transfert de la valeur depuis le stack DML vers une zone de travail qui serait traitée comme une variable de désignation.

A titre indicatif, cette boucle commencerait par initialiser un indice à la valeur 1 puis on aurait :

MOVE < OE > OF < FA > (indice) TO < VD > .

La fin de boucle comporterait l'incrémentation de cet indice et un saut en début de boucle.

Dans le cadre restreint du travail nous n'avons pas adopté cette solution et nous nous contentons de traiter les cas qui peuvent se ramener à une seule évaluation d'un test du langage cible.

Ainsi, le critère (: ADRESSE(: RUE = 'IMPASSE')) associé à FACULTE, donnera lieu à la génération suivante : IF RUE OF ADRESSE OF F1 = 'IMPASSE' ...

On se limitera dès lors aux quantificateurs qui suivent :

- si la relation qui aboutit à l'OE visé est de caractéristique J=1, les valeurs ALL, 1-, ## (elles auront le même sens);
- si la caractéristique J est supérieure à 1, sera seul admis i#.

CONCLUSION ET PROLONGEMENTS POSSIBLES

Dans ce travail, nous pensons avoir montré que le LDA est un langage suffisamment puissant pour permettre de structurer des programmes d'applications en boucles d'accès aux informations définies par le MSI.

L'analyse syntaxique du langage et la génération du code intermédiaire n'ont pas posé de problèmes particuliers, si ce n'est que, dans certains cas, le compilateur doit lire et retenir plusieurs symboles pour reconnaître certaines actions. C'est là une contrainte due à la volonté d'unifier la forme des différents types d'accès.

Nous avons rencontré plus de difficultés dans la génération du code objet. Ceci peut s'expliquer par la puissance du langage source ADL, mais surtout par la différence de conception qui réside entre ADL et le langage cible DML. Pour cette raison, le traitement des objets élémentaires a été réduit à sa plus simple expression.
Nous sommes arrivés à la conclusion que le meilleur moyen de réaliser le traitement complet des objets élémentaires était de rendre leur acquisition plus artificielle, de manière à unifier le traitement des objets élémentaires et des objets complexes. Ce nouveau point de vue nécessiterait une révision des algorithmes d'acquisition des objets complexes. Tel pourrait être un premier prolongement du travail, et on pourrait en profiter pour inclure l'implémentation des actions de modification.

Un autre prolongement possible est de confronter le langage aux réactions d'utilisateurs. Il serait peut-être souhaitable d'expliciter la sémantique de certaines actions en leur adjoignant des mots-clés. Nous pensons que l'action d'accès, dans sa forme étendue, constitue un bon compromis qui devrait rallier les suffrages d'une majorité.
Des utilisateurs pourraient encore reprocher au LDA la pauvreté de l'impression des résultats.

En dernier lieu, on pourrait s'attacher à rendre cette implémentation indépendante des systèmes cibles (le DML et SESAM, par transitivité). Les programmes objets seraient alors directement rédigés en COBOL et il faudrait aussi modifier le compilateur du DDL afin qu'il organise ses bases de données sur des fichiers accessibles par des programmes COBOL .

Annexe : syntaxe complète du LDA

< Programme > ::= < module > [< module >]$_{\infty}^{\circ}$

< module > ::= < séquence LH > | < bloc >

< bloc > ::= ENTER < nom racine > . < VD > $_{\circ}$< séquence > EXIT

< séquence > ::= < action > [; < action >]$_{\infty}$

< action > ::= < séquence LH > | < bloc > | < accès > | < accès fictif > | < action V >
< action OE > | < action élémentaire > | < addition > | < création R >
< suppression R > | < modification R > | < instruction conditionnelle >
< instruction GEN > | < ordre NEXT > | < ordre EXIT >

< accès > ::= { REACH / [} < relation > : < Q accès > < nom objet > . < VD > < condition >
< séquence > { END /] }

< accès fictif > ::= FOR < VD racine > [: < Q accès > < { < Vb > / VT } > . < VD > < condition > < séquence >]

< action V > ::= FOR { < VT > . < VD > / < VD > } < condition > < action >

< action OE > ::= < nom OE > < condition OE > { < création R > / < suppression R > / < accès > }

< action élémentaire > ::= PRINT | SE | SN | → < VT > | CLEAR < VT > | COMP < VT >

< addition > ::= A < nom OC > . < VD > < condition > → < VT >

< création R > ::= [C < nom relation > : < désignation >]

< suppression R > ::= [S < nom relation > : < désignation >]

< modification R > ::= [M < nom relation > : < désignation > | < désignation >]

< instruction conditionnelle > ::= IF < condition IF > THEN < séquence > ELSE
< séquence > END

< instruction GEN > ::= GENERATE < nom objet > . < VD > < séquence > END

< ordre NEXT > ::= NEXT < VD >

< ordre EXIT > ::= EXIT < VD >

< relation > ::= < relation * > [. < nom objet > . < relation * >]$_{\infty}^{\circ}$

< relation* > ::= < nom relation > *

< Q accès > ::= ALL | < ordinal >

< ordinal > ::= ## | < entier > # - { < entier > # / ## }

< condition OE > ::= = < valeur > [, < valeur >]$_{\circ}^{\infty}$ | ¬ { = / ≤ / ≥ } < valeur >

< valeur > ::= < nombre réel > | < chaîne de caractères > | < VD > * < VT >

< condition IF > ::= < terme IF > [/ < terme IF >]$_{\circ}^{\infty}$

< terme IF > ::= < facteur IF > [& < facteur IF >]$_{\circ}^{\infty}$

< facteur IF > ::= ¬ < critère appartenance > | ¬ < VD > < critère de relation >

¬ (< condition IF >)

< condition > ::= < terme > [/ < terme >]$_{\circ}^{\infty}$

< terme > ::= < facteur > [& < facteur >]$_{\circ}^{\infty}$

< facteur > ::= ¬ < critère appartenance > ¬ < critère relation > | ¬ (< condition >)

< critère appartenance > ::= $\left\{ \begin{array}{l} = \\ \end{array} \right.$ $\left\{ \begin{array}{l} < \text{valeur} > \\ < \text{désignation} > \end{array} \right.$ [, < valeur >]$_{\circ}^{\infty}$ | $\left\{ \begin{array}{l} \leq \\ \geq \end{array} \right.$ < valeur >

< critère relation > ::= (< relation > : < Q critère > $\left\{ \begin{array}{l} < \text{nom objet} > . < \text{VD} > \\ < \text{VT} > . < \text{VD} > \\ < \text{VD} > \end{array} \right.$ < condition >)

< Q critère > ::= < cardinal > | < ordinal >

< cardinal > ::= ALL | Ø | < entier > $\left\{ \begin{array}{l} - \\ + \end{array} \right.$ | < entier > - < entier >

< désignation > ::= $\left\{ \begin{array}{l} \text{ENTER} < \text{racine} > \\ \text{FOR} < \text{variable} > < \text{condition} > \\ < \text{nom OE} > < \text{condition} > \end{array} \right.$ < accès * >

< accès * > est un < accès > où la < séquence > ne comporte qu'une action qui est un < accès * > ou une < action V > , dont l'action est un < accès * >

REFERENCES ET BIBLIOGRAPHIE

[1] DEHENEFFE, HAINAUT, HENNEBERT, LECHARLIER, PAULUS
Système de conception et d'exploitation d'une base de données
Publication de l'Institut d'Informatique de Namur - 1974

[2] HAINAUT
Some tools for data independance in multilevel data base systems
IFIP TC-2 W.C. on Modelling in Data Base Management Systems - Nice - Jan. 1977

[3] *Interim report ANSI/X3/SPARC study group on data base management systems*
ANSI, Feb. 1975

[4] LALOUX
Implémentation du modèle d'accès par lui-même
Mémoire de fin d'études - Institut d'Informatique de Namur - Juin 1976

[5] HAINAUT, LECHARLIER
Présentation et spécification du modèle d'accès et du Langage Rigide
Documents internes - Institut d'Informatique de Namur - Février 1974

[6] HAINAUT, LECHARLIER
An extensible semantic model of data base and its Data Language
Proc. IFIP Congress 1974, North-Hollang publish. Co. 1974

[7] CHAMBERLIN, BOYCE
SEQUEL : A Structured English Query Language
IBM Research - RJ 1394 - 1974

[8] ABRIAL, BAS, BEAUME, HENNERON, MORIN, VIGLIANO
Projet Socrate
Institut de Mathématique Appliquée de Grenoble - 1970

[9] Colloque IRIA - Journées "Software des banques de données " (Aix-en-Provence)
IRIA - AFCET - 1970

[10] BASTIN
Etude des structures sémantiques communes aux systèmes IDS et IMS
Mémoire de fin d'études - Institut d'Informatique de Namur - 1975

[11] DATE
An introduction to data base systems
Addison - Wesley, 1975

[12] HENNEBERT
IMS/360
Documents internes - Institut d'Informatique de Namur

[13] HAINAUT
Evaluation des performances d'une base de données par modèle probabiliste
Séminaire INFORSID II - St Pierre de Chartreuse - Oct. 1976

[14] KNUTH
The art of computer programming
Vol 1 : Fundamental algorithms - Addison - Wesley, 1975

[15] GRIES
Compiler construction for digital computers
Ed. John Wiley & Sons, 1971

[16] HAINAUT
Transformation et évaluation d'expressions booléennes de conditions
Rapport technique - Institut d'Informatique de Namur - Fev. 1976

°
° °